DISS. ETH NO. 22298

Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

FLORIAN NEGELE

MSc ETH CS, ETH Zurich

born 15 August 1981

citizen of the Principality of Liechtenstein

accepted on the recommendation of

Prof. Dr. Jürg Gutknecht, examiner

Prof. Dr. Thomas Gross, co-examiner

Prof. Dr. Bernhard Egger, co-examiner

2014

Whatever can go wrong,
will go wrong.

(attributed to
Edward A. Murphy)

Murphy was an optimist.

(according to authors
of lock-free programs)

Acknowledgements

A dissertation is never the achievement of a single person. Now is the time to thank all people who enabled me to create and complete this thesis.

First and foremost I would like to express my special appreciation and sincere thanks to my good friend and advisor Dr. Felix Friedrich who tirelessly encouraged me for this work and always provided me with outstanding support and reliable expertise. It has been a great pleasure working with you all of these years.

I am also greatly indebted to my supervisor Prof. Jürg Gutknecht for giving me the opportunity and free reign to realise my ideas within this doctoral thesis. I am sincerely grateful for the comfortable working environment and his continuous guidance, help and patience. Additional thanks go to Prof. Thomas Gross and Prof. Bernhard Egger for their valuable assistance and kind agreement to co-examine this work.

I would also like to show my deep gratitude to Paul Reed who was always generous in scrutinising drafts and providing me steadily with helpful comments and much appreciated suggestions.

Last but not least, I would like to thank all of my past co-workers and colleagues for the interesting discussions, helpful insights, and the overall pleasant working environment. These people include Luc Bläser, Matthias Büchler, Olivier Clerc, André Fischer, Ulrike Glavitsch, Thomas Kägi, Daniel Keller, Sven Knudsen, Ling Liu, Roman Mitin, Georg Ofenbeck, Sven Stauber, Lars Widmer, and many others. I owe special thanks to Ruth Hidalgo, Franziska Mäder, and Martina Wirth for their reliable administrative support.

Contents

Abstract

The application of mutual exclusion in order to protect shared data from concurrent access is a recurring programming pattern found in contemporary application software. Although alternatives to blocking synchronisation like lock-free programming solve several critical problems and offer better progress guarantees, they have not become as popular. This especially applies to operating system kernels that provide support for multiprocessing environments.

This thesis describes our approach to design and implement an operating system kernel which is based solely on non-blocking algorithms. Its goal is to provide a completely lock-free runtime system for object-oriented programming languages that support concurrency and automatic memory management. We also strived for an uncompromisingly high portability of its source code and replaced all features provided by the hardware with machine-independent software solutions wherever possible. In particular, we abandoned the prevalent preemptive scheduler of modern systems in favour of a compiler-guaranteed cooperative multitasking. The combination of lock-free programming with cooperative multitasking presents a novel approach and enables several practical and beneficial non-blocking programming techniques such as processor-local storage.

The result of this work is a concise and reliable operating system kernel providing flexible language support for multithreading and garbage collection. Its source code is easily portable to a variety of hardware architectures ranging from powerful multicore machines to small embedded devices and microcontrollers. In addition, its rigorous machine independence also allows the runtime system to be used as a library for applications running on top of existing operating systems. Despite its simplicity, the performance of our lock-free approach is often comparable to established and optimised operating systems and is clearly superior to solutions based on locking.

Kurzfassung

Die heutige Softwareentwicklung setzt häufig Verfahren des wechselseitigen Ausschlusses ein, um gemeinsam genutzte Daten vor gleichzeitigem Zugriff nebenläufiger Prozesse zu schützen. Vielversprechende Alternativen zu dieser blockierenden Art der Prozesssynchronisation wie beispielsweise die lockfreie Programmierung lösen zahlreiche der damit verbundenen Probleme, sind aber längst nicht so weit verbreitet. Vor allem Betriebssystemkerne für Mehrprozessorsysteme bauen immer noch häufig auf gegenseitige Zugriffssperren.

Diese Arbeit geht dem Entwurf und der Entwicklung eines Betriebssystemkerns nach, welcher stattdessen ausschliesslich auf nicht blockierenden Algorithmen basiert. Das Ziel des Kerns ist es, ein durchgängig lockfreies Laufzeitsystem für objektorientierte Programmiersprachen mit Unterstützung für Nebenläufigkeit und automatischer Speicherbereinigung anzubieten. Ein weiteres Anliegen ist die kompromisslose Portabilität des Quellcodes, welche soweit möglich versucht, hardwarespezifische Funktionalitäten durch unabhängige Softwarelösungen zu ersetzen. Das bedeutendste Beispiel für diesen Ansatz stellt die Verwendung von kooperativem Multitasking dar, welches als Ersatz für das typischerweise präemptive Scheduling moderner Systeme zum Einsatz kommt. Die Verbindung von lockfreier Programmierung mit kooperativem Multitasking wurde bislang kaum ausgenutzt und ermöglicht einige praktische Programmiertechniken, die für die Entwicklung von nicht blockierenden Algorithmen von grossem Vorteil sind.

Das Ergebnis dieser Dissertation ist ein schlanker und zuverlässiger Betriebssystemkern mit flexibler Sprachunterstützung für Multithreading und Garbage Collection. Der Quellcode ist mühelos auf eine Vielzahl von Hardwarearchitekturen portierbar und läuft sowohl auf leistungsstarken Mehrkernrechnern als auch eingebetteten Systemen und Mikrocontrollern.

Die konsequent umgesetzte Maschinenunabhängigkeit erlaubt es dem Laufzeitsystem ausserdem, als Bibliothek für Applikationen bestehender Betriebssysteme zur Verfügung zu stehen. Die Auswertung der vorliegenden Arbeit hat gezeigt, dass die Leistungsfähigkeit unserer unkomplizierten und lockfreien Implementation in einigen Bereichen durchaus mit der von bewährten und optimierten Betriebssystemen vergleichbar ist. Dies gilt vor allen Dingen für Systeme, welche auf blockierender Prozesssynchronisation beruhen.

1 Introduction

This chapter describes the background, purpose, and scope of this thesis and puts it into perspective with related work. In addition, it summarises our main contributions by consolidating the ideas and concepts behind them.

1.1 Motivation

Since several years now, many operating systems provide native support for multiple central processing units in order to utilise the fullest potential of modern hardware. The most prominent representatives include for example Microsoft Windows and many Unix-like operating systems. Although the internal design and implementation of these operating systems often varies considerably, all of them usually include a generic hardware abstraction layer for applications running on top of them. The basic idea is to allow all applications to access system specific services and resources independently of the underlying hardware architecture.

The base at the lowest level of this abstraction mechanism consists of the so-called operating system kernel which manages and provides access to the most primitive but fundamental hardware resources such as logical processors, the main memory, and facilities to access peripheral devices. The functionality of the kernel is usually required directly or indirectly by all applications as well as higher-level components of the hardware abstraction layer like device drivers. In the context of a multiprocessing operating system however, all of the resources managed by a kernel are potentially used and shared amongst several applications and device drivers running in parallel on different processors. In this case, the implementation of an operating system kernel has to take special care in order to conduct and properly synchronise concurrent access to these shared resources.

One of the simplest and most popular forms of synchronising concurrent access to shared resources is a technique called mutual exclusion. It basically guarantees that only one of the interested parties has access to a resource at the same time. Modern operating systems usually provide a variety of different constructs that provide some sort of mutual exclusion. Examples include simple but efficient spinlocks or more sophisticated mutexes, semaphores, or monitors [HS08]. As a consequence, the use of mutual exclusion is very popular and prevalent in contemporary application software.

Despite its conceptual simplicity however, mutual exclusion and its various implementations have many well documented and understood drawbacks. For instance, mutual exclusion limits the progress of all contenders to a single one, effectively preventing any parallelism amongst them during the shared access. Since all other contenders have to wait for a single one, this mechanism is commonly known as blocking synchronisation. In fact, blocking synchronisation reveals many more deficiencies of mutual exclusion, most of which are described in more detail in Chapter 2. This is why alternative synchronisation primitives have been researched extensively since the early 1990s and still are. One particular outcome of these studies is the notion of lock-free programming which makes use of atomic operations while accessing shared resources concurrently. The basic goal of this approach is to allow several contenders to make progress at the same time.

Unfortunately, the results in this area have not yet been subject to widespread use, neither in application software nor in the implementation of the underlying systems. Operating system kernels in particular are usually still implemented using blocking synchronisation primitives. One example of a multiprocessor operating system employing this technique is called A2, a derivative of the Active Object System by Pieter Muller [Mul02] which was developed and maintained by the former Native Systems Group at ETH Zurich. Years of experience maintaining the A2 operating system have convinced us that adopting non-blocking synchronisation primitives reveals a great potential for simplifying and stabilising its kernel with respect to several recurring issues with its multiprocessor implementation.

Since its creation, the system and its kernel have been ported to several different hardware architectures and runtime environments. Each time, the source code had to be modified heavily in order to accommodate the peculiarities of the respective target hardware architecture [Egg01, Neg06]. Although the kernel was written in a high-level programming language, its source code was nowhere near as portable as desired, and it exposed many implicit hardware dependencies buried deep within its initial implementation. Until today, a lot of issues concerning the portability of the kernel have been identified and fixed, but we never truly achieved a streamlined unification of its many different versions in existence.

At the bottom of our thesis are the unsatisfactory shortcomings of the A2 operating system concerning its portability and multiprocessor implementation. Our main motivation for its improvement is based on the goal of creating a highly portable kernel on the one hand and the promising opportunities of non-blocking synchronisation in the context of operating systems on the other. The following sections draw an outline of our accomplishments.

1.2 Scope

Except for some minor assembly code sections, the Active Object System as well as its newest incarnation A2 are written in a high-level programming language called Active Oberon [Rea04]. This is a backward compatible extension of Niklaus Wirth's Oberon programming language [Wir88] and one of the latest instalments in the family of Pascal based languages as depicted in Figure 1.1 on the following page. The most important additions are the support for a modern style of object-oriented programming and the notion of active objects for concurrency [Gut97]. Active objects are objects that integrate a lightweight process known as a thread or activity. These three terms will be used interchangeably throughout this thesis.

The kernel of the A2 operating system provides a complete runtime system for the Active Oberon programming language. This runtime system includes the support for modules with dynamic loading, active objects in a

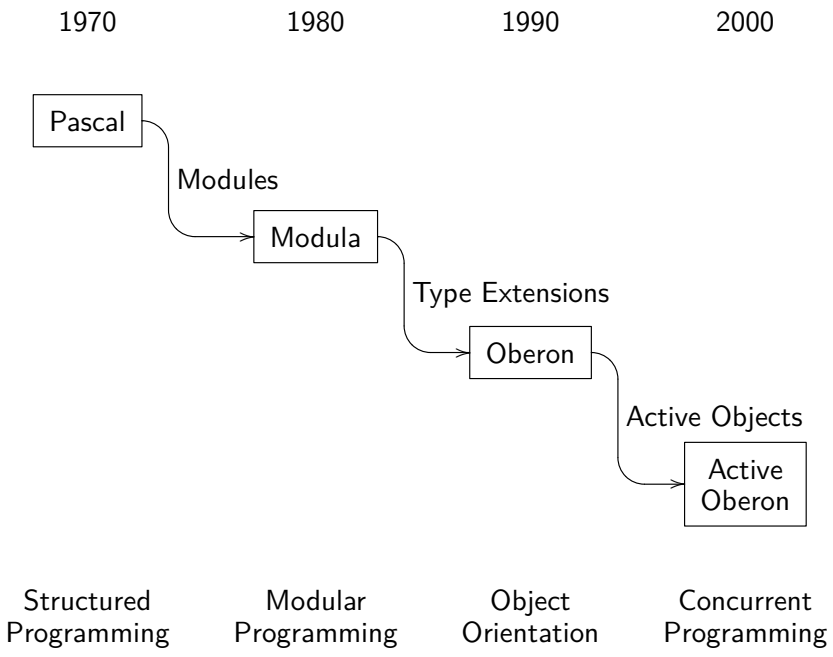

Figure 1.1: Descendants in the family of Pascal based programming languages and their extensions. Adapted from the Active Oberon language report by Patrik Reali [Rea04].

multiprocessing environment, and an automatic memory management using a garbage collector based on mark and sweep. In addition, the kernel also provides a low-level framework for accessing and controlling peripheral hardware [Mul02].

The scope of this thesis is partially defined by the core functionality of the original A2 operating system kernel. However, its design and implementation endeavours to achieve an uncompromisingly high portability of its source code using only lock-free programming techniques. The innovation comes from imagining of how all of the basic system services can be implemented having these objectives in mind. We call the resulting implementation a runtime system rather than a kernel of an operating system as it is designed to fulfil two different purposes. One the one hand, it still features a scheduler for executing activities on multiple processors and a memory manager including a conditionally supported garbage collector. The runtime system can therefore be used on a machine without an underlying operating system, in which case the system additionally supports the handling of interrupts and provides low-level access to peripheral hardware. As a result, the runtime system can be used as an in-place replacement for the original A2 kernel as it mimics all of its functionality.

On the other hand however, our work is also designed to be the basic runtime system for applications that are executed on top of existing operating systems like Microsoft Windows or systems based on the Linux kernel. If an Active Oberon program is compiled targeting one of these runtime environments, the runtime system is just an additional application library linked into the resulting binary executable file. In this case, the runtime system cannot access the hardware resources directly because they are usually not available to standard applications. Instead, the runtime system makes use of the system interface provided by the operating system itself. For example, the scheduler of the runtime system has no direct access to the processors of the system. It mimics their functionality by creating lightweight processes or threads that act as actual processors. The same technique is applied to the memory manager, where all memory allocations are just forwarded to the corresponding application programming interface of the operating system.

All of these replacements are enabled by a very small and concise abstraction layer inside the runtime system itself consisting only of a handful of functions. These functions are replaced automatically by the linker with their environmental specific substitutes. The underlying execution environment of the runtime system is therefore completely transparent to the application running on top of it. This kind of source code portability allows us to target different runtime environments as well as hardware architectures easily and without much effort.

Although the runtime system is written in and provides support for the Active Oberon programming language, the concepts and ideas presented in this thesis are not limited to this language in any way. The runtime system provides a generic abstraction for memory management, lightweight processes, and interrupt handling. This is the fundamental basis for an operating system kernel and could be implemented just as well in any other system programming language. The most important aspect of the language invariance however is the tight coupling of the runtime system with the compiler. In contrast to lower level programming languages that can translate most of their constructs directly into machine code, Active Oberon features some sophisticated mechanisms which require the assistance of a runtime system. Therefore, the Active Oberon compiler generates a lot of meta data and implicit calls to functions provided by the runtime system. This implies that all code has to be compiled by the same compiler in order to conform to the programming interface provided by the runtime system. If another language is to be used for implementing the runtime system, the compiler used for generating the executables must be modified accordingly. It is therefore necessary for programmers to have full control over the implementation of the programming language. The simplicity of Active Oberon and its implementation was one reason why we opted for this particular programming language.

Another basis for this decision was the high level of abstraction provided by Active Oberon as well as its strong type system which guarantees many runtime checks. Other system programming languages like C or C++ on the other hand, are notorious for their broad range of unchecked runtime errors caused by misusing their features. In C and C++ parlance, this sort of errors

is known as undefined behaviour. In many cases, the robustness of modern operating systems stems from astounding efforts that have been undertaken to cope with undefined behaviour. Especially where shared resources like the main memory are concerned, these systems have a strong tendency to make use of hardware facilities whenever possible in order to detect and handle faulty processes.

An example is the implementation of virtual memory which is often provided by dedicated memory management units. Virtual memory can be used to protect an operating system from erroneous and corrupting memory accesses but render its implementation inherently non-portable. Active Oberon as implemented by the compiler and the runtime system provides all of these checks by itself and does therefore not require any virtual memory management. This approach effectively reduces the necessary hardware support and achieves a high portability across different architectures.

1.3 Related Work

Using non-blocking algorithms and data structures for implementing multiprocessor operating systems has been investigated since over twenty years now. Massalin and Pu were amongst the earliest to deliver a non-blocking implementation of an operating system kernel [MP91]. The kernel of their multiprocessor operating system called Synthesis included support for threads and virtual memory as well as a file system [Mas92]. They showed that operating system kernels using non-blocking synchronisation are practical and achieve at least the same performance as conventional systems. Similar conclusions have later been confirmed many times, for example by Greenwald and Cheriton [GC96, Gre99].

However, the implementations of the resulting non-blocking operating system kernels relied on an atomic double compare-and-swap operation called DCAS. This operation is an extended version of the more common single compare-and-swap operation known as CAS which allows to atomically compare and exchange the values of two unrelated memory locations instead of one. Based on their results, Greenwald and Cheriton argued that

this operation is sufficient for practical non-blocking operating systems. Unfortunately, the hardware support for this particular operation is very limited and most contemporary hardware architectures do not provide it at all.

The single compare-and-swap operation on the other hand is supported widely on shared-memory architectures capable of multiprocessing. This may be due to the postulations of Herlihy who showed early on that single atomic read-modify-write operations are the most general and can even substitute other atomic operations [Her91]. For portability reasons, this thesis relies only on the single compare-and-swap operation in order to achieve the broadest hardware support available. There are several other implementations of non-blocking operating systems that followed the very same approach. Hohmuth and Härtig for example focused on non-blocking real-time systems by utilising only the single compare-and-swap operation in order to improve portability [HH01].

However, most of the non-blocking kernels in the literature are based on Unix-like environments and are therefore written in popular system programming languages like C or C++. In comparison to Active Oberon, these programming languages often depend on the underlying operating system in order to detect faulty processes. Our approach on the other hand relies completely on runtime checks provided by the implementation of the programming language. As a consequence, the corresponding runtime environment established in this thesis is many times simpler and smaller than the systems targeted in related work.

The lock-free runtime system presented in this dissertation makes use of several non-blocking data structures and algorithms. Most of them have already been investigated extensively in the standard literature. For instance, there exist many lock-free implementations of abstract data structures like queues or lists [Val94, MS96, HLM03]. Examples of non-blocking algorithms as well as complete software components include lock-free memory managers and garbage collectors [HM91, Mic04c, GGH07]. For what it is worth, all of this research is designed throughout to be applicable in programs running on on top of modern operating systems in order to be most useful.

However, lock-free algorithms and data structures are always implicitly based on and restricted by the characteristics of the underlying runtime environment. Considering modern operating systems, the corresponding execution environment is most often implemented using preemptive multitasking. This means that any lock-free algorithm has to be prepared that a context switch to another task could potentially happen at any time. To the best of our knowledge, lock-free programming has hardly been considered in contexts that use cooperative multitasking instead. In contrast to the standard literature, this thesis explores the advantages and consequences of lock-free programming in execution environments where task switches are under the control of the lock-free algorithm itself. All lock-free data structures and algorithms presented in the following chapters are considered under this novel point of view.

1.4 Contributions

The main result of this work is a portable multiprocessor operating system kernel and runtime system for the Active Oberon language. Its implementation does not rely on blocking synchronisation and takes advantage of the novel approach of combining lock-free programming with cooperative multitasking. The runtime system itself is based on the following contributions of this thesis:

1. We propose a set of fundamental programming techniques for authors of lock-free programs. The techniques are enabled by cooperative multitasking and are designed to simplify the implementation of non-blocking algorithms and to reason about their progress.

2. Applying these techniques, we provide a lock-free implementation of an unbounded concurrent queue that is especially well suited for being integrated into task schedulers. In contrast to other lock-free queues in the literature, our approach does not unconditionally allocate memory in enqueue operations which renders task switches inexpensive.

3. We present a scalable scheduler for cooperative multitasking of lightweight processes on multiple processors with shared memory. It supports interrupt handling and provides a lock-free framework for implementing sophisticated synchronisation primitives on top of it.
4. We provide a lock-free implementation of a memory manager for memory heaps of arbitrary sizes. In addition, we complement the manual memory management with a non-blocking garbage collector which is designed to be an ordinary and optional process instead of an integral part of the runtime system.

1.5 Overview

The contents of this dissertation are organised in the following chapters:

Chapter 2 focuses on the concepts of lock-free programming in general and introduces memory models and programming techniques which are required by the subsequent chapters.

Chapter 3 exemplifies solutions to recurring problems of lock-free programming by stepwise refining an implementation of a lock-free queue which is used extensively by the runtime system.

Chapter 4 presents the design and implementation of a lock-free scheduler which is used for implicit cooperative multitasking on multiple processors.

Chapter 5 describes the design and implementation of a lock-free memory manager including a concurrent and incremental garbage collector.

Chapter 6 discusses several case studies concerning the portability of the runtime system and summarises the necessary software abstractions.

Chapter 7 evaluates the performance of various system components and compares them with related work.

Chapter 8 concludes this thesis and identifies future directions.

2 Lock-Free Programming

This chapter introduces the concept of lock-free programming in general and the definitions associated with it. After the specification of a simple memory model, it presents some useful programming techniques for implementations of practical lock-free programs.

2.1 Introduction

The term lock-free programming is used to describe the application of non-blocking atomic operations instead of blocking synchronisation primitives provided by a scheduler. Herlihy and Shavit provide an excellent overview of multiprocessor programming in general that includes both synchronisation techniques [HS08].

The most common usage of blocking synchronisation primitives is to achieve mutual exclusion of processes or threads competing for a shared resource. Other applications include the notification of arbitrary events or conditions being met. Blocking synchronisation is in general characterised by the fact that there are potentially one or more processes waiting for another process to progress. This form of synchronisation is ubiquitous in concurrent programs and most modern multiprocessor operating systems often support a variety of different blocking synchronisation primitives usually known as locks, mutexes, semaphores, or monitors. Similarly, almost all programming languages with built-in support for concurrency also provide some constructs for blocking synchronisation.

Despite its simplicity and availability however, blocking synchronisation traditionally suffers from several well-known problems. Generally speaking, all of them are based on the absence of a guarantee about how long a blocked process has to wait. The following list summarises the most common issues:

- Deadlocks occur whenever a group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed.
- Livelocks happen when competing processes are able to detect a potential deadlock but make no observable progress while trying to resolve it.
- Starvation characterises the repeated but unsuccessful attempt of a recently unblocked process to continue its execution.

In general, waiting processes depend heavily on the cooperativeness of competing processes. If, for whatever reason, a process fails to unblock waiting processes, they can remain blocked forever. There exist several solutions and recommended programming patterns that try to cope with all these concrete problems. However, there are also many principal challenges conditioned by the fundamental design of blocking synchronisation. For instance, mutual exclusion prevents any parallelism among competing processes by definition. Even if there is no contention at all, the setup for mutual exclusion adds overhead for each access to a shared resource. This can be compensated to some extent by increasing the amount of shared data that is protected by the same synchronisation primitive. On the other hand, a more coarse-grained locking scheme prevents processes from accessing shared but potentially independent data at the same time. As a consequence, there is some trade-off that has to be considered.

In addition, it is often complicated to actually prove that the usage of the primitives is correct and that the issues raised above cannot happen. In the context of operating system kernels for example, blocking synchronisation is especially problematic when interrupts are used to handle external events. An interrupt causes a processor to temporarily suspend the execution of the currently running code in order to complete an interrupt handler instead. The problem arises when the handler needs to access shared resources that are currently in use by the interrupted code and have therefore already been protected using mutual exclusion. In this case, the mutual exclusive acquisition of the resource has to be made insensitive to the potential

reentrancy in order to ensure that the interrupt handler can still proceed successfully. Achieving reentrant blocking synchronisation is difficult but so crucial that failure to do so has negative impacts on the reliability and responsiveness of the complete operating system.

The term non-blocking synchronisation and its research have their seeds in the dissatisfaction with mutual exclusion and its implications. Herlihy was one of the first to coin the term in the early 1990s [Her90]. In the beginning, the term lock-free was used synonymously with non-blocking and implied that a concurrent algorithm does not require any blocking synchronisation primitives like locks at all. Instead, a non-blocking algorithm relies solely on atomic operations provided by the hardware architecture that allow to change a memory location in a transactional manner. The most common operations are test-and-set, fetch-and-add, load-link and store-conditional, or the superior compare-and-swap operation. Herlihy showed that the latter is universal and can therefore actually implement all other atomic operations [Her91].

From the beginning, the goal of non-blocking synchronisation was to devise some basic guarantees about the progress of non-blocking algorithms. An algorithm is said to make progress if the number of steps it takes to complete its operation has an upper bound [Her91]. Since the early 2000s, the following three different basic notions of non-blocking progress are distinguished [HLM03]:

- Obstruction-freedom guarantees that an algorithm makes progress even if its operation was obstructed previously by other algorithms. This implies that algorithms do make progress when run in isolation, but not necessarily when other algorithms are executed concurrently. This is the weakest non-blocking property.

- Lock-freedom guarantees that at least one algorithm makes progress even if other algorithms run concurrently. This non-blocking property ensures system-wide progress and therefore implies obstruction-freedom. However, it also implicates that some algorithms may starve forever or require an undetermined number of steps to proceed.

- Wait-freedom guarantees that all algorithms eventually make progress. This is the strongest non-blocking property and combines lock-freedom with starvation-freedom. It typically relies on algorithms being aware of the progress of others and cooperatively helping them if necessary.

Hence, the term lock-free nowadays describes a property of non-blocking algorithms rather than the absence of blocking synchronisation primitives. Herlihy has shown that any algorithm can be implemented wait-free if enough resources like memory and processing power are available [Her88]. This is a rather theoretical statement since designing wait-free algorithms is a very complex task and only a few practical and efficient wait-free algorithms are known [KP12]. Lock-freedom on the other hand is much easier to achieve and often considered to deliver wait-free progress in practice. It has been proven, that a large class of lock-free algorithms are de facto wait-free and can be closely bounded in the average number of steps they take to complete [ACHS14]. All algorithms presented in this thesis are lock-free by design and we will show how cooperative multitasking helps to improve the progress of contending processes.

Despite its progress guarantees, lock-free programming should not be considered to be the holy grail, as it has its own set of challenges. First of all, non-blocking algorithms are in general very hard to get right. As if the implementation of concurrent programs was not difficult enough, the lack of a proper process synchronisation renders non-blocking algorithms highly non-deterministic. Programmers must always keep in mind that in the worst case, the state of the complete program could have been changed after the execution of a single instruction. As a consequence, even marginal modifications of the source code can lead to a completely different behaviour at runtime. This severely impacts all sorts of debugging facilities and provokes so-called heisenbugs which change their behaviour as soon as they are observed.

Furthermore, there is the omnipresent ABA problem which describes a category of situations when a process fails to recognise that the state of the program has been changed in the meantime. Since non-blocking

synchronisation uses atomic read-modify-write operations, the state of a program is always stored and encoded in a few small shared memory locations. Without further precaution, a process erroneously assumes that the program state is still the same if the values of these memory locations have not changed since the last readout. This can lead to disastrous consequences like corrupted data structures if any other process concurrently modified the corresponding memory locations meanwhile.

Nevertheless, we will show how to deal with all of these kinds of problems by providing useful programming techniques for practical lock-free algorithms in the last part of this chapter. But first we have to specify what actually constitutes an atomic operation.

2.2 Memory Models

Many recent programming languages with support for concurrency define a so-called memory model. A memory model describes the behaviour of concurrent programs that share data between their threads of execution. A careful design of a memory model enables many compiler optimisations while still providing important guarantees for the programmer. Java was the first programming language whose specification included an explicit and extensive description of its underlying memory model [GJSB05]. Its first release was generally considered flawed and had to be revised several times. Over the time, other popular languages like C or C++ have also evolved into multithreaded programming languages. Their technical standards have been updated in order to incorporate the notion of a memory model as well [ISO11a, ISO11b].

The Active Oberon programming language used for the implementation of this thesis currently still lacks this kind of specification [Rea04]. Due to the missing guarantees for programmers, there exist many programs written in Active Oberon whose authors obviously acted on implicit assumptions about the interaction of active objects without proper synchronisation. As a consequence, the resulting programs are often non-portable because they highly depend on the implementation chosen by the compiler.

One of our goals was to overcome this unsatisfying situation and to establish a solid foundation for the lock-free algorithms as presented in this thesis. In this section, we will dissect some of the most popular memory models of other programming languages in order to derive a concise specification of a simple yet powerful and portable memory model suitable for Active Oberon. In contrast to often overly complex memory model definitions, our approach aims at staying true to the nature and heritage of the Active Oberon programming language by striving for simplicity.

2.2.1 Properties of Common Memory Models

Today, almost all of the most widely used programming languages provide some notion of concurrency support. Typically, the concept of concurrent execution of code is accompanied by built-in facilities that enable synchronisation of concurrent executions. This kind of synchronisation is required when concurrent threads of execution share memory or other resources. Objected oriented programming languages like Java or C# for example provide special language constructs that try to solve this issue with mutual exclusion using locks and monitors [GJSB05, ECM06]. Additionally, the libraries provided by these programming languages typically feature advanced synchronisation constructs that enable more fine-grained control over critical sections.

Often however, programming languages allow programmers to access shared data without enforcing any proper synchronisation. In this case, reading and writing shared data across different threads has to be done very carefully because it may lead to unpredictable or at least unexpected behaviour. In order to still provide some basic guarantees for programmers, a precise memory model is required to describe the behaviour of shared data in this situation. In absence of such a specification, programmers should ideally not rely on unsynchronised access to shared resources. Programs that ignore this implication inherently assume some sort of behaviour during their execution and are therefore likely to behave differently if compiled using a different compiler or if executed on another platform or execution environment.

Another advantage of specifying a memory model for programming languages is to enable implementations to optimise multithreaded programs. In this context, one of the most prominent optimisation techniques is out-of-order execution which allows to decrease the performance penalties of memory accesses. Reordering instructions is often applied by the compiler since it can deduce data dependencies in a program directly from its source code. In the case of Java and C# for example which both run inside a virtual machine, the code can be rearranged once again just before it gets executed. Finally, some implementations of hardware architectures such as the 32-bit and 64-bit architectures by Intel and AMD are also able to reorder and cache memory accesses on a per instruction level basis [Int14, Adv13]. A memory model should not restrict these different stages of optimisation which are able to improve the efficiency of multithreaded programs substantially.

For the following discussion of common properties we will focus on the specifications of the memory models of the Java and the C# programming language [GJSB05, ECM06]. Although the memory models share many equivalent characteristics, they differ significantly in the effort made to describe them. The current edition of the Java programming language for example describes the underlying memory model at great length. The detailed specification is obviously targeting an audience of compiler developers and may be confusing and overwhelming for ordinary users of the language. The specification of C# on the other hand does not mention the actual term memory model at all.

- Both programming languages use the concept of synchronisation barriers to describe the operation of their synchronisation constructs. Synchronisation barriers are sequence points in the execution of a program where all changes of one thread become accessible to another. All built-in synchronisation primitives like synchronised methods or lock statements act as synchronisation barriers. Once the mutual exclusion is released and transferred to a different thread, all changes become accessible instantaneously. In general, synchronisation barriers constitute the necessary guarantees that allow programmers to reason about the state of shared data at any given point in time.

- Java as well as C# define volatile fields which are declarations of field variables of fundamental type that include a modifier called volatile. Both memory models specify the ordering of read and write accesses to volatile fields with respect to any other memory references in the instruction sequence of multithreaded programs. Reading a volatile field for example is guaranteed to happen before any other memory reference in the instruction sequence of one thread, while writing a volatile field will happen after any other memory reference preceding the instruction sequence. The process of accessing a volatile field can therefore also be regarded as a synchronisation barrier.

- Apart from proper synchronisation constructs provided by the languages and their libraries, both specifications do not provide any other guarantees. In particular, memory references to non-volatile fields may happen in arbitrary order as long as their effect equals the observable behaviour of the instruction sequence as executed by a single thread. The overall purpose of this notion of a sequential execution order is to explicitly enable optimisations like reordering instructions on any stage of the execution environment. This intentionally includes the compiler, the runtime system, as well as the hardware architecture. At the same time however, the memory models still provide a hardware abstraction that allows programs to be portable across different platforms.

2.2.2 A Memory Model for Active Oberon

Based on the previous analysis we derived a memory model suitable for lock-free programming using the Active Oberon programming language. Although our proposal implements a memory model for this language, our approach is in general not bound to any particular programming language and is applicable to many execution environments. Deriving a specification for Active Oberon provided an excellent opportunity to define the mechanics of memory accesses beyond synchronised constructs many programs in existence relied upon.

```
procedure CAS(reference to variable, old, new)
    value ← variable
    if value = old then
        variable ← new
    return value
```

Listing 2.1: The function of the atomic compare-and-swap operation.

The original Active Oberon language report only included the description of how active objects are synchronised and how shared objects can be protected from concurrent access [Rea04]. All object instances integrate a monitor that provides support for mutual exclusion and awaiting conditions. Similar to the corresponding facilities found in Java or C#, the operations on this monitor naturally act as synchronisation barriers. Unfortunately, the language report does not mention what happens if shared memory is accessed without having been protected properly beforehand. This kind of concurrent memory access is actually crucial for lock-free programming where algorithms and access to data structures are intentionally not synchronised.

Non-blocking algorithms need support for atomic operations to access shared memory. The powerfulness of the atomic compare-and-swap operation combined with its wide availability on modern processors was the main reason why we wanted to include it in the Active Oberon programming language. We extended the original language specification of Active Oberon by introducing a predefined compare-and-swap operation called CAS, defined in Section C.1 on page 204. The function of this built-in procedure is shown in Listing 2.1. It takes three arguments of the same basic or reference type, where the first argument indicates the shared variable to be changed. First, it compares the value of the shared variable with the value of the second argument. If the two values match, the variable is overwritten with the value of the third argument. The return value of the procedure is the original value of the variable. The whole operation is executed atomically and its result is visible instantaneously to other processes.

If the second argument matches the third one, the operation does not have to actually write the shared variable and is conceptually equivalent to an atomic read of the variable. This operation is necessary for consistency on platforms that perform ordinary read operations on single memory locations by executing two or more consecutive instructions. Another advantage of this atomic operation is the fact that we can require compilers to implement it to be a synchronisation barrier in addition to the synchronisation constructs provided by Active Oberon.

While defining a memory model for Active Oberon we kept the tradition of that programming language and its predecessors in mind. Our focus was to keep the memory model as simple as possible but still provide a mechanism for lock-free programming without sacrificing any potential optimisations. The resulting memory model implemented by our compilers for Active Oberon consists of the following two simple rules:

1. Data that is shared between two or more activities at the same time has to be protected using exclusive blocks unless the data is read or modified using the atomic compare-and-swap operation.

2. Changes to shared data become accessible to other activities after leaving an exclusive block or executing an atomic compare-and-swap operation. Implementations are free to reorder all other memory accesses as long as their effect equals a sequential execution within a single activity.

This memory model is comparably simple but not restrictive and explicitly allows compiler and hardware optimisations. It does not rely on the definition of volatile variables but still provides portable means for concurrent access. On the other hand, it provides important guarantees for programmers concerning the visibility and consistency of modified shared data. This is an important requirement for authors of lock-free algorithms in order to reason about the correctness of their programs. Therefore, this memory model is sufficient to provide a reasonable foundation for lock-free programming which is used extensively in this thesis.

2.3 Programming Techniques

In this section, we will present a set of useful programming techniques for developing lock-free algorithms. Each of them is exemplified by a practical application written in the Active Oberon programming language [Rea04]. All techniques are based on the underlying cooperative scheduler that will be described in detail in Chapter 4.

In contrast to preemptive multitasking which finds a use in almost all modern operating systems, cooperative multitasking requires all participating processes to cooperate with each other in order to guarantee system-wide progress. Cooperative processes are basically coroutines which take on the responsibility of passing the control of execution on to other processes [Con63]. In the context of non-blocking synchronisation however, cooperative multitasking not only requires but actually also allows a process to decide for itself when it will yield execution to another process. This fine-grained control offers several advantages over preemptive scheduling where task switches occur asynchronously and are therefore completely transparent to the non-blocking algorithm. All algorithms and data structures presented in this thesis are based on combining lock-free programming with this advantage of cooperative multitasking. The fundamental idea is to completely omit and delay task switches during the execution of a non-blocking algorithm.

2.3.1 Disabling Task Switches

In our implementation of the Active Oberon language, all task switches required by cooperative multitasking are managed by the compiler instead of the application. The compiler inserts the necessary task switches at various places in the program. The exact mechanism and the actual insertion points are described in detail in Chapter 4. The advantages of this approach which we call implicit cooperative multitasking are twofold. First of all, the compiler is able to automatically insert all task switches that are necessary in order to ensure the cooperativeness of all participating processes which basically guarantees system-wide progress.

Second, all task switches are transparent to the programmer and do not require any user intervention. As a consequence, the source code of a program does not need any modifications regardless of whether it is executed under preemptive or cooperative multitasking. However, we explicitly allow programmers to temporarily disable the automatic insertion of task switches by the compiler. The corresponding Active Oberon language feature we have introduced is called an uncooperative block. Uncooperative blocks are compound block statements with a special modifier called uncooperative, defined in Section C.12 on page 214.

The semantics of an uncooperative block enforce the compiler to completely elide the generation of any implicit task switches while translating the statement sequence encompassed by the block statement. This rule only applies to the immediate code sequence of one particular block and is not transitive to any other statement sequence executed indirectly by calling a procedure for example. The reason is to make the cooperativeness of a statement block a compile-time property of the source code rather than an expensive policy that has to be checked at runtime.

Each process executing the statement sequence of an uncooperative block does not implicitly yield execution to another process unless task switches are requested explicitly by the programmer. Although technically possible, we highly recommend programmers to refrain from doing so, since it defies the whole purpose of uncooperative blocks. For the further discussion, we therefore assume that there are neither implicit nor explicit task switches in an uncooperative block.

We call the statement block uncooperative because each process executes the encompassed statement sequence without stopping as soon as it has entered the block and is never interrupted by any other process until it leaves the block. This has the nice implication that on a machine with a single processing unit, only one process at a time actually executes an uncooperative block. In this particular case, the contained statement sequence is effectively executed under mutual exclusion. As a consequence, uncooperative blocks can be regarded as synchronisation primitives which interestingly enough do neither use any sort of blocking nor incur any runtime overhead whatsoever.

The most important property of uncooperative blocks however is the fact that the number of concurrently executing processes is also limited even on a machine with more than one processor. Because of the absence of any task switches, each processor has to execute an uncooperative block in its entirety. Thus, a process is neither interrupted by another process nor executed partially on a different processor. On hardware with several processing units, the maximal number of processes concurrently executing an uncooperative block is therefore always bounded by the actual number of physical processors. The fact that this number has an architectural limit and is always statically known is one of the main ideas of this thesis and we try to exploit it wherever possible.

The advantages of having a statically known upper bound of processes concurrently executing an uncooperative block are manifold. All of them are usually not available using a preemptive scheduler and have therefore hardly been considered in practice:

1. The upper bound allows to make better estimates about the progress even under high contention. When a process enters an uncooperative block and hits the architectural limit, it is guaranteed that all processors are busy executing either the same or another uncooperative block. Before any other process can enter the uncooperative block, one or more currently running processes must have left some other block beforehand.

   In comparison, in a preemptive scheduler the number of processes executing arbitrary statement blocks are in general completely unbounded. In principle, a block could be entered by any number of processes without requiring a single one of them to actually proceed.

2. Since there are no task switches during the execution of an uncooperative block, the actual time spent in a block depends only on the actual statement sequence and the number of contending processors. Regarding a preempted statement block, this execution time also depends on the total number of processes in the system which can cause arbitrarily long delays.

The statement sequence within an uncooperative block is typically quite short and is always executed by the same processor. Thus, in order for a processor to repeatedly starve, it would have to be constantly overtaken by contending processors. For all practical purposes, starvation is therefore in general much less likely than using preemptive multitasking.

3. Lock-free algorithms sometimes need information about all currently contending processes. This applies especially to processes executing wait-free algorithms which require the state of other processes in order to help them finishing their operation in time. Since preemptive schedulers permit any number of contending processes, the required information has to be associated with each process and gathering it can be difficult and expensive especially since all temporarily suspended processes have to be considered as well.

   Our approach effectively reduces the number of contending processes down to the amount of processing units which is usually a much smaller number. In addition, the maximal value of this number is always known at compile time which allows to store the required information in static arrays. This allows us to associate the requested information efficiently and readily available for other processes with the algorithm itself. We will exemplify this programming technique in Section 2.3.3.

Regarding preemptive multitasking, the same results could technically speaking also be achieved by temporarily disabling preemption for the duration of the uncooperative block. In the typical case where preemption is implemented using interrupts, this would imply that the corresponding interrupt handling mechanism must be disabled as well. Although turning interrupts off is an inexpensive operation, it can severely impact the stability, responsiveness, and reliability of the complete system. This is why the control over interrupt handling is in practice most often secured by the operating system and usually not accessible to ordinary applications running on top of it.

Using cooperative multitasking on the other hand, task switches do not rely on hardware facilities like interrupts and can therefore be implemented completely in software. There is no need to enable or disable interrupts during the execution of an uncooperative block and it does usually not matter if they are enabled or not. The only exceptional case is the fact that in the course of handling an interrupt, the processor might execute the very same uncooperative block that was interrupted beforehand. In the worst case, the uncooperative block could even be executed by several nested interrupt handlers at once. Of course, the uncooperative block cannot be considered to be executed under mutual exclusion on a single processor machine in this scenario.

Although interrupt handlers are executed on physical processors, they can conceptually also be regarded as being processed by an additional virtual processor that only runs every once in a while. Hardware architectures often explicitly allow and are designed to handle several nested interrupts at once. But the nesting level itself is always limited to the number of different interrupt handlers, since a handler should not interrupt itself. Therefore, the maximal number of virtual processors is equal to the architectural limit of nested interrupts. Thus, the sum of physical and virtual processors has always an upper bound and is defined by the hardware architecture and known for any given system. Consequently, the conclusion of always having a limited set of processes executing any uncooperative block still holds even with enabled interrupt handling.

The ability of representing interrupt handlers as virtual processors also helps to reason about the problematic reentrancy in interrupt handlers. Since lock-free programming intentionally omits blocking synchronisation primitives, a non-blocking algorithm can potentially be executed by any number of processes instead of one at a time. As long as the algorithm does not rely on any specific information related to the currently running process or processor, it is completely irrelevant which processes or processors actually execute the algorithm concurrently. From the perspective of the algorithm, it could theoretically even be executed concurrently by the same process or processor at the same time. Therefore, non-blocking algorithms are usually considered to be reentrant by design.

```
procedure INCREMENT(reference to counter)
    uncooperative                                              ▷ (1)
        repeat
            previous ← CAS(counter, 0, 0)                      ▷ (2)
            value ← CAS(counter, previous, previous + 1)       ▷ (3)
        until value = previous                                 ▷ (4)
        return previous
```

Listing 2.2: An example of a lock-free operation incrementing a shared counter variable and returning its previous value.

With the help of virtual processors we can represent an interrupt handler to be executed by an additional process running on a distinct processing unit. As a result, the issue of reentrancy of uncooperative blocks completely vanishes. Our approach even works when non-blocking algorithms require information about the process or processor during their execution. In short, we solve this problem by assigning the same basic context information to a virtual processor that is also required by an ordinary process. The actual interrupt handling applied in this thesis is described in detail in Section 4.5.

2.3.2 Basic Lock-Free Operations

An example of one of the simplest but still useful and practical non-blocking data structures is an atomic counter. Its operations allow to atomically increment and decrement the integer value of the counter. Considering modern hardware architectures, the increment operation is sometimes provided natively by the processor itself. Unfortunately, atomic increment instructions do usually not allow to consistently determine the actually incremented value. This is for example valuable when contending processes require unique integer values.

Listing 2.2 shows a sample implementation of a portable increment operation on an atomic counter which returns its previous value in a consistent manner. It follows the basic structure of all lock-free algorithms:

1. The whole body of the procedure is marked as uncooperative causing the compiler to omit task switches completely while compiling the procedure. This guarantees that there is an upper limit of contending processes potentially executing this operation in parallel which reduces the probability of starving as discussed above.

2. The value of the shared variable is read atomically using a compare-and-swap operation and stored in a local copy. This particular invocation of the compare-and-swap operation does not modify the variable but it is applied in order to ensure that the value of the whole variable is read atomically. This is important for accessing shared and concurrently modified variables consistently on architectures where ordinary reads and writes require more than one hardware instruction.

3. A subsequent compare-and-swap operation tries to atomically modify the shared variable based on its value read beforehand. Even if this operation is called by several processes, one compare-and-swap operation has to be the first according to the sequential ordering defined by the memory model. As a consequence, there is always one process that will succeed even under high contention rendering the whole operation lock-free.

4. If the compare-and-swap operation returns a different value than expected, one or more other processes were faster and changed the variable already. In this case, the increment operation must be repeated by reading the variable and trying to modify it again.

The increment operation is lock-free but not wait-free because a single process can theoretically starve when it never succeeds in the third step. This can only happen when one or more different processes repeatedly succeed modifying the value of the shared counter in the meantime. However, each process that does succeed also exits the uncooperative block and returns from the procedure. Since the number of processes doing so is limited by the number of processors, there are no additional processes involved which would cause the initial process to starve.

Thanks to cooperative multitasking, starvation can only be provoked when the participating processors modify the same shared counter over and over in a tight loop. In practice however, it is completely unrealistic that a single processor is forever outperformed by contenders even in this pathological case [ACHS14]. For example, the number of instructions required for repeatedly executing the whole procedure is much larger than the relative short instruction sequence for atomically reading and modifying the shared variable. Moreover, in contrast to preemptive multitasking, the instruction sequence is never interrupted by any other process. If the instruction sequence would have been preempted, other processes could cause arbitrarily long delays and even successfully modify the shared counter in the meantime. Therefore, the ability to disable task switches in lock-free algorithms renders unsuccessful atomic compare-and-swap operations much more unlikely.

2.3.3 Using Processor-Local Storage

Modern programming languages and operating systems alike support the notion of thread-local storage which allows to access global data that is unique to the currently running thread. This facility is typically implemented by instantiating a unique copy of a predefined set of global variables for each new thread and associating it accordingly. The main advantage of thread-local storage is that these variables are still globally accessible but behave like local variables and do therefore not require any access synchronisation. A completely different usage can be found in wait-free algorithms which often make use of thread-local storage to store and publish the current state in order to allow other threads to help them making progress.

However, thread-local memory associated with a thread is usually not designed to be shared across multiple threads as it is for example reclaimed as soon as the thread stops its execution. In addition, the usage of thread-local storage incurs a performance overhead when accessing thread-local variables or creating threads. Moreover, the corresponding memory may always be allocated upon the creation of the thread regardless of whether the latter actually uses some of its thread-local storage or not.

Using cooperative multitasking, we were able to improve upon these issues. The key idea is to gather all accesses to thread-local variables inside uncooperative blocks. Since there are no task switches while executing an uncooperative block, any thread-local variable is always accessed by the same processor during that period. The required set of distinct instantiations of global variables for concurrent access is therefore limited by the number of processors in the system. We call that set processor-local storage as it consists of one copy of predefined global variables per processor rather than per process. This technique offers several advantages:

1. The amount of memory allocated for processor-local storage does not depend on the number of processes in the system. Its size is only limited by the actual number of processors. Since the upper bound of this number is statically known, processor-local storage can be simply represented by global arrays where each element is associated with one individual processor.

2. Using distinct elements of an array, processor-local storage can be efficiently accessed using the unique index of the currently running processor. As before, accessing the array does not need any synchronisation since each processor only modifies its own element. Furthermore, the creation of a process does not incur any overhead as all processor-local data can be allocated statically at the beginning of a program.

3. Using an array, the set of data local to all processors can be defined together with the actual algorithm which requires that data. The algorithm can therefore easily query all processor-local data that is concurrently used by other processors by traversing the corresponding array.

In general, thread-local storage might still be useful and cannot be replaced completely by our approach. For example, there is no guarantee that two consecutive uncooperative blocks are executed by the same processor because an intermediate task switch could cause one process to be resumed

by another processor. Therefore, it is impossible to cache data specific to the currently running processor while executing two consecutive uncooperative blocks. The index of the current processor should only be used to refer to data that is associated with that particular processor since the actual value of the index typically varies from the viewpoint of a single process. However, our approach is sufficient for all of our purposes and we think that it provides a more elegant solution than its counterpart. Additionally, all of our algorithms can actually be implemented without requiring the runtime system or extending the programming language to support the notion of thread-local storage.

A practical application of processor-local storage is improving the performance of the compare-and-swap operation under high contention. Since atomic operations like compare-and-swap act as synchronisation barriers, they usually put a strain on the caches of multiprocessor memory devices. Their actual performance drops all the more under high contention because there are more atomic memory accesses that have to be synchronised. In the case of non-blocking algorithms where failed operations are typically repeated until they succeed, memory devices tend to congest even more. As a result, atomic operations have to be considered several times more expensive than standard machine instructions. See Figure 7.1 on page 147 for the actual performance of the compare-and-swap operation under increasing contention on a concrete machine with 32 processors.

In order to cope with the performance degradation of atomic compare-and-swap operations under high contention, Dice et al. have analysed several so-called contention management algorithms [DHM13]. The basic idea behind contention management is to delay the repetition of failed atomic operations in order to prevent memory devices from congesting. Dice et al. identified a technique called exponential backoff to be one of the best performing contention management algorithms on contemporary hardware architectures. This algorithm keeps track of how many times the compare-and-swap operations failed and waits for an exponentially long time if that number hits a machine-dependent threshold. The actual waiting time depends on the number of failures as well as on some algorithm specific parameters that have to be tuned carefully for each machine.

More important than the actual details of the waiting time in this context however, is the fact that Dice et al. utilise thread-local storage in order to count the failures. This is convenient but conceptually flawed since the congestion is not triggered by threads but rather the underlying processors running the threads. Using thread-local storage, newly created processes always start out with the same initial value of the failure counter. This number may need some time first to stabilise at a certain level which might distort the actually required waiting time during that period. Regarding a system with a preemptive scheduler, it is possible for a task switch to occur while a process is waiting because of a failed compare-and-swap operation. If the scheduler resumes a process that will execute an atomic operation as well, the corresponding processor did not wait enough time between the two atomic operations and may cause the memory device to congest. In addition, the original process may be resumed at a time when there is no congestion any more causing it to continue its wait for too long.

Listing 2.3 on the following page shows a sample implementation of an atomic compare-and-swap operation which makes use of processor-local storage in order to implement exponential backoff for N processors. Each processing unit uses its unique index exemplarily called *processor* to access its own element of a global array counting the failures. In comparison to the implementation by Dice et al., the number of failures is therefore associated with each processor instead of each process. This effectively solves all issues raised above since the actual number of failures is always up to date regardless of which process has actually executed the atomic operation. In addition, the surrounding uncooperative block ensures that there are no task switches while waiting. As a consequence, the corresponding processor will always wait for the necessary time after an atomic operation failed regardless of how many concurrent processes there are.

Processor-local storage and all other programming techniques presented in this section are used extensively in the lock-free algorithms discussed in the remainder of this thesis. The following chapter for example discusses a lock-free implementation of a concurrent queue data structure which makes use of processor-local storage in order to implement safe memory deallocation.

```
variable
    failures : array N of integer

procedure CompareAndSwap(reference to variable, old, new)
    uncooperative
        value ← CAS(variable, old, new)
        if value = old then
            if failures[processor] > 0 then
                failures[processor] ← failures[processor] − 1
        else
            failures[processor] ← failures[processor] + 1
            if failures[processor] > threshold then
                Wait(2^min(failures[processor]·factor,maximum))
        return value
```

Listing 2.3: Compare-and-swap operation using an exponential backoff contention management algorithm for N processors. Adapted from Dice et al. [DHM13].

3 Case Study: Lock-Free Queues

This chapter acts as a case study of lock-free programming and describes the implementation of a concurrent and unbounded first-in first-out queue data structure. We use queues extensively within the scheduler of our runtime system in order to keep track of the set of all activities that are currently not running. This chapter illustrates several development stages by stepwise improving and revising a naive implementation into a practical, safe, and memory-efficient final version. Along the way, it also reveals several typical pitfalls of lock-free programming and proposes generic solutions to them.

3.1 Introduction

Queues are fundamental and popular data structures which are used in a variety of contexts, ranging from simple software applications to the implementation of the underlying operating systems. Their field of application can also include multiprocessing environments where data structures are potentially accessed concurrently by several processes. In this case, the corresponding operations of the data structures have to be properly synchronised in order to guarantee their correctness. Because of their popularity and broad field of application, the synchronisation of queues has been researched thoroughly in the standard literature.

Besides many implementations based on mutual exclusion, also several non-blocking versions of concurrent queues have been proposed. Hwang and Briggs for example belong to the earliest researchers who have presented a non-blocking queue based on the atomic compare-and-swap operation [HB84]. Further lock-free implementations include the studies by Mellor-Crummey [Mel87], Herlihy [Her90, HLM03], Massalin and Pu [MP91], and Valois [Val94].

However, the single most popular lock-free queue implementation is probably the one devised by Michael and Scott [MS96]. There are several reasons why their queue set a standard which is widely adopted in the literature as well as in practice. By drawing ideas from the work by Valois, their implementation is simple and one of the fastest to date [HS08, KP11]. In contrast to some of the algorithms proposed by other researchers, their lock-free queue implementation is highly practical because it explicitly allows empty queues as well as an arbitrary number of concurrent enqueue and dequeue operations. In addition, unlike some other methods, it does not require an atomic double compare-and-swap operation which is hardly supported by contemporary hardware.

All of the unbounded lock-free queues in the standard literature including the one by Michael and Scott are based on representing queue items using a singly linked list. The list consists of a linked chain of intermediate nodes which do not store the actual user data but only a reference to it. As a consequence, each operation that inserts a new element into the linked list also allocates memory for the intermediate node representing that element.

There are also wait-free queue implementations, for example the one by Kogan and Petrank [KP12]. They guarantee starvation-freedom by publishing information about each atomic operation a process is about to perform on a queue in order to allow contending processes to help each other. Unfortunately, these algorithms therefore not only allocate memory per element but also for each of the atomic steps which accumulate to a total of three memory allocations per enqueue operation.

The allocation of memory seems to be ubiquitously accepted as a necessary means to implement non-blocking data structures. But nevertheless, the overhead of allocating memory may actually make up the largest part of a lock-free operation and we are surprised that this issue has never been addressed before. Regarding the task scheduler of our runtime system for instance, each task switch implies that the currently running activity gets suspended and therefore placed into a queue. We believe that this simple operation is critical and should ideally not require any memory allocation at all. Otherwise, a basic task switch could lead to a system failure if there is no more memory available, or even trigger a full garbage collection.

As a consequence, we refrained from applying an existing solution and tried to come up with an implementation of a lock-free queue that is more suitable for our purposes. We based our first approach on a basic and unsynchronised algorithm that satisfies our memory constraints. Listing 3.1 on the next page shows a standard implementation of an unsynchronised queue representing its elements by a singly linked list. It consists of a basic data structure called QUEUE which stores references to the first and last elements of the linked list. A single element is represented by a data structure called ITEM which just stores a reference to the next element in the list. This data structure is intended to be a base type which has to be extended by users of the queue in order to associate it with meaningful user data. In our scheduler for example, we extend the representation of an activity from this base type and are therefore able to enqueue and dequeue activities into any queue.

The technique of representing elements of a container using base types requires the programming language to support user-defined type extensions. It can therefore be often found in programs written in object-oriented programming languages. For example, it has been applied extensively in the implementation of Project Oberon [WG92]. An advantage of this technique is its generality, as it allows users of the queue data structure to store arbitrary data per element in the linked list. But most importantly, it chains the elements directly rather than indirectly using intermediate nodes which are typically used as place holders for the actual user data. Therefore, once an element is allocated by the user, it can be enqueued and dequeued repeatedly without requiring additional memory allocations. This is the main reason why we chose this representation as the base for our lock-free queue implementation.

An obvious restriction of this technique however is the fact that user data as represented by a single item may not be part of two distinct queues at the same time. If an element is currently part of a queue, it has to be dequeued first before it can be enqueued again into the same or another queue. This limitation poses no problem in our scheduler, since activities are either currently running, or enqueued exclusively in a single queue in order to be resumed.

```
structure ITEM
    next : pointer to ITEM

structure QUEUE
    first : pointer to ITEM
    last : pointer to ITEM

procedure ENQUEUE(item, queue)
    item→next ← null
    if queue.last ≠ null then
        queue.last→next ← item
    else
        queue.first ← item
    queue.last ← item

procedure DEQUEUE(queue)
    item ← queue.first
    if item ≠ null then
        queue.first ← item→next
        if queue.first = null then
            queue.last ← null
    return item
```

Listing 3.1: Data structures and operations of an unsynchronised implementation of an unbounded first-in first-out queue.

The queue implementation in Listing 3.1 on the following page is not safe to be used by several activities in parallel, as both of its operations are not protected properly from concurrent access. If there are no additional operations accessing the underlying linked list, it would actually be sufficient to protect both procedures simply by applying mutual exclusion in the form of a blocking synchronisation primitive. This would signify the actual code of both operations as critical sections which may only be executed by one process at a time. In order to achieve a non-blocking alternative to this approach however, the algorithm has to be rewritten significantly. But first, we have to identify the set of all algorithmic steps which appear to contending processes to take effect instantaneously.

3.2 Linearisation Points

Our memory model specifies that any change to a shared variable becomes accessible to other activities immediately after the execution of the atomic compare-and-swap operation. In general, if an operation appears to take effect instantaneously at some point in time between its invocation and response, it is called linearisable. The notion of linearisability was devised by Herlihy and applies to all atomic operations [HW90]. The atomic step when a linearisable operation takes effect is generally known as its linearisation point. According to our memory model, executing a modifying compare-and-swap operation as well as leaving an exclusive block both serve as linearisation points.

In the absence of proper synchronisation of concurrent activities that execute non-blocking algorithms, any atomic change of a shared variable is immediately visible to other activities. By implication, the state of shared variables must always be considered to be volatile in between the execution of atomic operations. In the worst case, any possible state can and will eventually be observed by an activity and it is the task of the programmer to reason about the correctness of the non-blocking algorithm in all of these states. Identifying the set of linearisation points of an algorithm allows to isolate the state space that has to be considered for this purpose.

Listing 3.2 on page 39 shows our first approach to implement a lock-free queue. We naively tried to mimic the implementation of the unsynchronised version and used therefore the same data structures as before. Complying with the memory model as defined in Section 2.2.2, we replaced all accesses to potentially shared variables in both procedures of the unsynchronised version with atomic compare-and-swap operations. This pertains to the references of the first and last queue element as well as to the successor of the last queue element. In addition, we reordered statements and added loops in order to take account of failed compare-and-swap operations according to the guidelines given in Section 2.3.2. Although this version might look correct at first glance, its implementation is severely flawed.

The linearisation points in our first approach include all modifying compare-and-swap operations and are labelled accordingly in Listing 3.2. Both procedures have three distinct linearisation points which can be summarised as follows:

E_1: The ENQUEUE procedure repeatedly tries to swing the reference to the last element of the queue to the item given as argument. The successor of this element is guaranteed to be invalid as it is reset at the beginning of the procedure.

E_2: If there previously was a last element, the reference to its successor is set to the new item. This operation must succeed because any other activity enqueueing an element operates on a different reference to the last element.

E_3: If there was no last element, the queue was empty and the reference to its first element is changed instead. This operation might fail if another element was enqueued in the meantime. However, this case can be ignored because the reference to the first element would then already point to the correct item in the linked list.

D_1: The DEQUEUE procedure returns with a special value if there is no first element. This linearisation point does not modify shared data but still takes effect immediately from the point of view of the caller.

```
procedure ENQUEUE(item, queue)
    item→next ← null
    repeat
        last ← CAS(queue.last, null, null)
    until CAS(queue.last, last, item) = last                    ▷ E1
    if last ≠ null then
        CAS(last→next, null, item)                              ▷ E2
    else
        CAS(queue.first, null, item)                            ▷ E3

procedure DEQUEUE(queue)
    repeat
        first ← CAS(queue.first, null, null)
        if first = null then
            return null                                         ▷ D1
        next ← CAS(first→next, null, null)
    until CAS(queue.first, first, next) = first                 ▷ D2
    if next = null then
        CAS(queue.last, first, null)                            ▷ D3
    return first
```

Listing 3.2: Operations of a naive approach to implement a lock-free queue.

D_2: If the queue is not empty, the procedure tries to swing the reference to the first element of the queue to its immediate successor. If this operation was successful, the previously first element of the queue is now dequeued. Otherwise, another activity already modified the queue in which case the dequeue operation must be repeated all over.

D_3: If the dequeued element has no successor, the queue has been emptied. In order to ensure the correctness of the next enqueue operation, the reference to the last element of the queue has to be invalidated if it matches the dequeued element.

Although the sequential execution of both procedures is obviously correct in isolation, there are several issues when they are executed concurrently. For the sake of the argument, we assume that there are only two activities P and Q concurrently accessing a single queue. Using the linearisation points identified above, we can easily detect execution sequences that lead to a corruption of the linked list. For this purpose, it is sufficient to look at a few execution sequences where the linearisation points of both procedures are interspersed with each other.

First, we look at an execution sequence where activity P calls the ENQUEUE procedure and enqueues a new element called A. Right after P reaches linearisation point E_1, denoted by $P|E_1$, the other activity Q invokes the DEQUEUE procedure. Activity P continues after Q has reached its first linearisation point. Figure 3.1 on the facing page shows the states of two different queues in this scenario. On the left hand side, the initial queue is empty and the corresponding sequence of reached linearisation points is $P|E_1, Q|D_1, P|E_3$. Activity Q leaves the DEQUEUE procedure immediately after reaching linearisation point D_1 since the reference to the first element has not yet been modified by P and is still invalid. Activity P can therefore complete its operation without interruption and leaves behind a correct data structure.

The other queue depicted on the right hand side of Figure 3.1 on the next page initially contains a single element called B. Since the queue is not empty, different linearisation points are reached and the corresponding sequence is $P|E_1, Q|D_2, P|E_2$. Here, activity Q resets the reference to the first element of the queue because B has no successor yet. Although the successor is just about to be set when P reaches E_2, the queue data structure is now corrupted because the reference to the first element does not point to the actual first element A. Any subsequent dequeue operation returns prematurely after reaching D_1 while enqueue operations append elements without ever correcting the reference to the first element.

This problem does not occur if the initial queue contains more than one element. In this case, the first element already has a valid successor and Q is able to correctly modify its reference after reaching D_2. When P reaches E_2, it eventually sets the missing successor yielding a valid linked list.

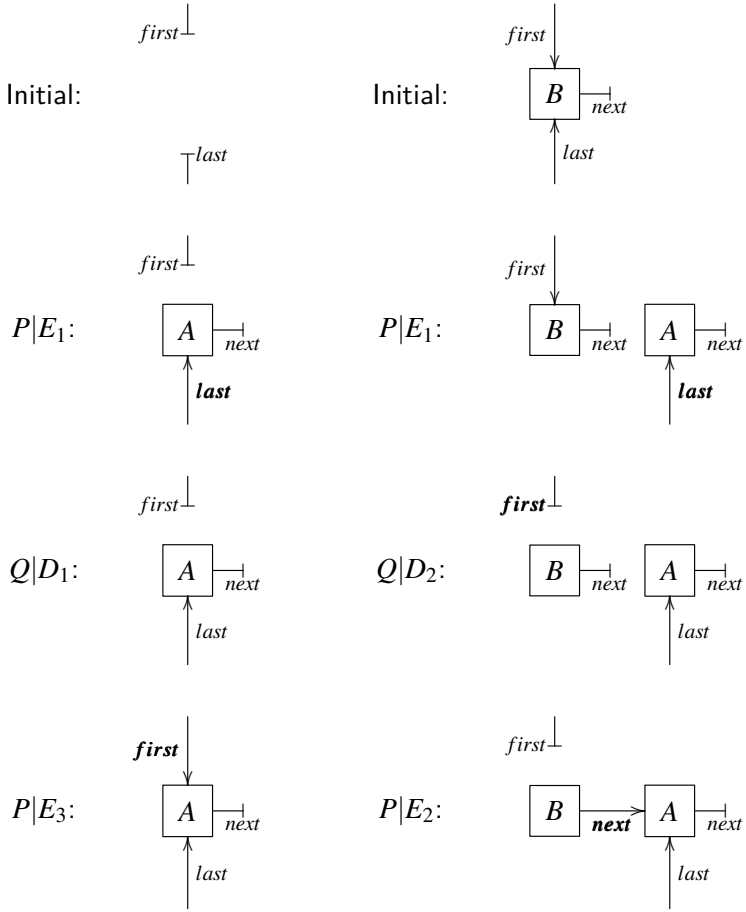

Figure 3.1: States of two different shared queues after reaching linearisation points while one activity P enqueues an element and another activity Q dequeues an element.

Another sample execution sequence describes a similar scenario where a dequeue operation is interspersed with a concurrent enqueue operation instead. The corresponding sequence of linearisation points on a shared queue containing a single element is $Q|D_2, P|E_1, Q|D_3, P|E_2$. In this case, activity Q resets the reference to the first element of the queue when reaching linearisation point D_2 as before. This time however, activity P fails to properly readjust this reference because it reaches linearisation point E_2 instead of E_3 in the end. As a consequence, the resulting queue gets corrupted the same way as described above.

A third example illustrates the case where one activity gets suspended in between two subsequent linearisation points of an operation but is never resumed. Although this scenario is unlikely in practice since it involves some sort of external events which must interrupt the activity within an extremely short time frame, it has to be nevertheless considered in all non-blocking algorithms. Regarding the ENQUEUE procedure, any activity that fails to reach linearisation points E_2 or E_3 does not connect the enqueued element to the linked list. Although subsequent enqueue operations can successfully append new elements, the linked list remains partitioned into two separate parts. The same applies to activities that are suspended in between linearisation points D_2 and D_3 and therefore fail to update the reference to the last element of a queue that got empty. In this case, a subsequent enqueue operation does not update the reference to the first element of the queue because the reference to its last element has not been invalidated. As a consequence, enqueue operations succeed while any dequeue operation aborts prematurely in D_1.

Besides proving that our algorithm is incorrect, the linearisation points also help to reason about its progress guarantees. Since both procedures contain a single loop, it is sufficient to show in which cases they may loop forever. Both operations fail to make progress when they never reach linearisation points E_1 and D_2 respectively. In both cases, the linearisation points coincide with modifying compare-and-swap operations. Therefore, these atomic operations must fail repeatedly in order to cause an activity to get stuck permanently in the corresponding loop. Although this starving is very unlikely in practice, the algorithm cannot be wait-free by definition.

However, even if an activity actually starves it has not yet successfully executed an atomic compare-and-swap operation. Hence, there has not been any change visible to other concurrently running activities. The starving activity can therefore never obstruct the operation of other activities. On the other hand, the starvation of one activity implies that there is at least one other activity that successfully executed an atomic compare-and-swap operation. As a result, the successful activity always makes progress by exiting the corresponding loop and renders the algorithm therefore lock-free. We will use the same sort of argumentation for all other lock-free algorithms in this thesis.

We could try to rewrite our lock-free procedures in order to fix all the issues raised above. In fact, we tried out several variations and each time reordered the linearisation points inside the procedures. However, the inherent problem with our algorithm does not lie in its operations but in the choice of the underlying data structure which forces us to treat empty queues differently from non-empty ones.

The advantage of having two references to each end is that both queue operations can ideally operate on their respective part without interfering with each other. But whenever an enqueue operation inserts an element into an empty queue it also has to update the reference to the first element of the queue. Conversely, a dequeue operation removing the last remaining element of a queue has also to invalidate the reference to its last element. Since we are using compare-and-swap operations that modify single memory locations, only one of both references can be updated at a time. As a consequence, if two references have to be modified simultaneously, only one of them can be up to date in between their modifications. The other reference must be deemed to be lagging behind and all procedures have to prepared for this issue.

Since we have only two references that have to be modified successively in two different procedures, there are at most four possible combinations of how to choose the most current reference in each case. In our naive implementation for example, we chose the reference to the last element to be the most current in the ENQUEUE procedure whereas the first element is always up to date in the DEQUEUE procedure. For each possible

combination however, we have found sequences of linearisation points that eventually lead to a corruption of the linked list. Consequently, this data structure has to be abandoned in favour of a design that does not have to update two references in both procedures.

An alternative approach consists of discarding the reference to the last element altogether. This solution does not suffer from the problems described above since there is only one reference to keep up to date. However, it requires enqueue operations to actually traverse the linked list in order to get to the currently last element of the queue. The obvious decrease in performance of querying the last element going from constant time to linear complexity is unfortunate at least. However, a real problem occurs when enqueue operations traversing the linked list are overtaken by concurrent dequeue operations. Without further precaution, any dequeued element that is immediately enqueued into another queue could cause a traversal to be potentially proceeded in a different linked list. We did not pursue this approach further since solving the intricate problem of concurrently traversing elements seems to be disproportionate to its actual value.

The next section follows another idea and unifies the representation of empty and non-empty queues instead. It describes an improved version of our algorithm as well as the consequences of incorporating the notion of a sentinel element.

3.3 Intermediate Nodes

Our next version of the queue data structure introduces a distinguished dummy element of the linked list called the sentinel. The sentinel is preallocated and always the first element in the linked list regardless of how many actual items there are. Its sole purpose is to unify the representation of empty queues and non-empty queues. An empty queue consists only of the sentinel element and is referenced by both pointers to the first and the last element of the queue. Both references therefore always point to an existing element and it is never necessary to invalidate them while enqueueing or dequeueing an item.

```
structure ITEM

structure NODE
    next : pointer to NODE                  ▷ successor in linked list
    item : pointer to ITEM                  ▷ associated element

structure QUEUE
    first : pointer to NODE                 ▷ first node of linked list, sentinel
    last : pointer to NODE                  ▷ last node of linked list, may lag behind
```

Listing 3.3: Data structures of an unbounded first-in first-out queue making use of intermediate nodes.

Regarding our initial design, we still want users of the queue to provide their own user data in the form of an extension of the basic ITEM data structure. In a non-empty queue, the sentinel element therefore points to its successor which stores the actual user data of the first item. If a dequeue operation advances the reference to the first element to its successor, the latter automatically becomes the new sentinel. Although the new sentinel holds the actual user data that has been dequeued, it is still part of the linked list and therefore cannot be returned to the user. Otherwise, the user could immediately enqueue this item in the same or another queue breaking the linked list of the original queue.

This problem can be solved by adding another level of indirection in the form of an intermediate data structure called NODE that references the actual enqueued item. The linked list therefore consists of a chain of nodes where the first is the sentinel node and all others have the actual item associated with them. Listing 3.3 shows an improved definition of the data structures which have been modified accordingly. Although this design solves the problem raised above, it unfortunately also subverts our goal of memory-efficiency since intermediate nodes require memory allocations. For the sake of the argument, we will ignore this issue for now and resolve it in a later section.

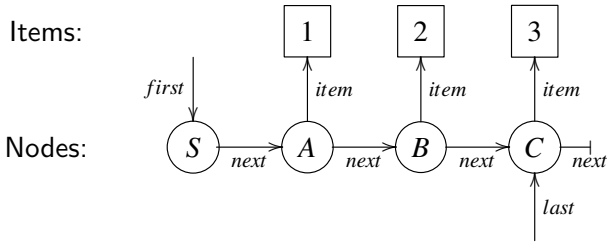

Figure 3.2: Example of a queue containing three elements referenced by a linked list of four nodes.

Figure 3.2 depicts a sample instantiation of this queue data structure. It consists of a linked list of four nodes *S*, *A*, *B*, and *C* where the first one is the distinguished sentinel node. While *S* has no item associated with it, the actual queue items numbered from one to three are referenced by the nodes *A*, *B*, and *C* respectively.

The invariant of this data structure is that the linked list of nodes is always up to date. The list always begins with the sentinel node and ends with the first node that has no valid successor. The reference to the last node always points to some node of the linked list and is intentionally allowed to lag behind.

The idea to this design was drawn from Michael and Scott [MS96] who in turn based their work on Valois [Val94]. The advantage of its invariant is that the reference to the last node does not have to be up to date since the last node can always be identified by a missing successor. Furthermore, thanks to the introduction of a sentinel node, enqueue operations do not need to modify the reference to the first node any more in order to indicate an empty queue. As a result, the enqueue operation only operates on the successor of the last node in order to keep the linked list up to date. The reference to the last element can be used to find the actually last node efficiently but does not need to be updated at the same time.

```
procedure ENQUEUE(item, queue)
    node ← ALLOCATE()
    node→next ← null
    node→item ← item
    repeat
        last ← CAS(queue.last, null, null)
        next ← CAS(last→next, null, node)                ▷ E1
        if next ≠ null then
            CAS(queue.last, last, next)                  ▷ E2
    until next = null
    CAS(queue.last, last, node)                          ▷ E3
```

Listing 3.4: Enqueue operation making use of intermediate nodes.

Listing 3.4 shows the lock-free enqueue operation for the improved queue data structure. Apart from some minor rearrangements, it corresponds to the algorithms presented by Valois [Val94] and Michael and Scott [MS96]. In this preliminary version, the enqueue operation always allocates a new intermediate node at the beginning. For the sake of convenience, we assume for now that all allocated nodes occupy different memory regions.

As required by the invariant of the data structure, the ENQUEUE procedure updates the reference to the last node after modifying the linked list. The corresponding linearisation points are labelled in the listing and can be summarised as follows:

E_1: The enqueue operation first tries to update the successor of the currently last node. This operation fails if another enqueue operation has overtaken it at this step causing the corresponding reference to the last node to lag behind.

E_2: If there is already a successor, the reference to the last node obviously lags behind. In this case the procedure tries to swing the reference to the last node to its successor in order to repeat the enqueue operation on a subsequent node of the linked list.

```
procedure DEQUEUE(queue)
    repeat
        first ← CAS(queue.first, null, null)
        next ← CAS(first→next, null, null)
        if next = null then
            return null                                  ▷ D1
        CAS(queue.last, first, next)                     ▷ D2
        item ← next→item
    until CAS(queue.first, first, next) = first          ▷ D3
    FREE(first)
    return item
```

Listing 3.5: Dequeue operation making use of intermediate nodes.

E_3: If the modification of the successor was successful, the procedure finally tries to update the reference to the last node by swinging it to the currently appended node. This operation fails if this reference already lags behind again at this point.

The corresponding dequeue operation is shown in Listing 3.5 and has the following linearisation points:

D_1: As before, the dequeue operation returns with a special value if the sentinel node has no successor indicating an empty queue.

D_2: If the reference to the last node points to the sentinel, the procedure advances it to its immediate successor before removing the sentinel from the linked list. Otherwise, the reference to the last element points to a node which is not part of the linked list after the removal.

D_3: The procedure tries to swing the reference to the first element of the queue to its immediate successor. This operation fails and has to be repeated if another dequeue operation as overtaken it in the meantime. Beforehand, the item associated with the node that is about to become the new sentinel is stored as return value.

The correctness of this algorithm has been shown by Michael and Scott in terms of safety and liveness [MS96]. Here, we deduce its correctness by arguing that there is no sequence of linearisation points reached by concurrent activities that violates the invariant of the data structure. Without loss of generality, we will focus on all combinations of single linearisation points sequentially reached by two concurrent activities P and Q operating on a single shared queue. Sequences of linearisation points involving more than two activities can always be pairwise reduced to this set of combinations and must therefore be correct by induction. Both conditions of the invariant can be considered separately:

1. The invariant requires the linked list of nodes to be chained correctly at all times. The linked list consists of the sentinel node and all of its direct and indirect successors. The corresponding linearisation points potentially modifying the linked list are therefore only E_1 and D_3.

   If both activities reach the same linearisation point, only the corresponding compare-and-swap operation of the first activity in the sequence can succeed because both try to modify the same memory location. In the sequence $P|E_1, Q|E_1$ for example, only P will succeed leaving behind a correctly linked list, while Q will have to repeat the operation because the successor has been updated in the meantime. The same applies to the sequence $P|D_3, Q|D_3$ where only P will succeed in advancing the sentinel node, while Q has to repeat the loop because the reference to the first node no longer matches its expected value. In both cases, only one activity modified the data structure and did not violate the invariant while doing so.

   Regarding the sequences $P|E_1, Q|D_3$ and $P|D_3, Q|E_1$, both of the corresponding compare-and-swap operations are potentially able to succeed because they modify two distinct memory locations in each case. If for example the compare-and-swap operation in $P|E_1$ was successful, then P has modified the reference to the next node of either the sentinel or one of its successors. The sentinel node will therefore always have a valid successor and Q cannot violate the invariant when advancing the reference to the first node in D_3.

Regarding the second sequence $P|D_3, Q|E_1$ on the other hand, the only conceivable way to break the invariant is when Q modifies the successor of the previous sentinel node which was just removed from the linked list by P. But the corresponding compare-and-swap operation in E_1 cannot succeed, since the previous sentinel node must have had a successor or else P could not have reached D_3 in the first place.

2. The invariant additionally requires the reference to the last node always to point to a node that is part of the linked list. The linearisation points that potentially modify this reference are E_2, E_3 and D_2. In E_2 and D_2, the reference the last node is modified to point to its immediate successor. If the atomic compare-and-swap operation in E_3 succeeds, the reference now points to the currently appended node. Therefore, the reference to the last node always points to one of its direct or indirect successors when modified.

   It remains to show that the reference to the last node does not lag too far behind. The only way this could happen is when this reference is overtaken by the reference to the first node while advancing the latter in D_3. However, this linearisation point is always preceded by D_2 which ensures that the reference to the first node and the last node are never the same before removing the sentinel. Hence, the reference to the last node always points to one of the successors of a potentially removed sentinel.

As in the previous section, the linearisation points also allow to derive the progress guarantees of the algorithm. Both procedures contain a single loop which could hinder an activity from proceeding. In order for an activity to get stuck permanently in one of these loops, the compare-and-swap operations at E_1 or D_3 must fail repeatedly. Since both operations try to modify distinct memory locations, there must have been another activity that actually succeeded while executing the same compare-and-swap operation. As a consequence, there exists at least one activity that actually does proceed even if others starve executing the same queue operation.

During the execution of any of the two loops, each activity additionally updates the reference to the last node in case it falls behind. The corresponding linearisation points are E_2 and D_2. Both can help an ongoing enqueue operation to advance the reference to the appended node in E_3. This technique of helping concurrent activities is very common in lock-free programming. The general idea is to satisfy preconditions oneself instead of waiting for another activity to establish them. The same technique even allows to compensate for queue operations that never reach their last linearisation point because they are suspended and never resumed for whatever reason. The algorithm is therefore also correct when activities halt halfway in between two consecutive linearisation points. Concurrent activities can therefore not obstruct each other and the algorithm is therefore lock-free by definition.

At this point, we have improved our first queue implementation and turned it into a correct lock-free algorithm. But unfortunately, we also sacrificed the desired memory-efficiency along the way. The next section will get rid of our oversimplifications and discuss the consequences of reusing nodes in order to save memory allocations wherever possible.

3.4 The ABA Problem

The next iteration of our lock-free queue algorithm reintroduces the same kind of memory-efficiency our very first approach implemented. The basic idea is to pair each item that is currently not enqueued with a node that can be reused for that purpose whenever the item is enqueued again.

As discussed above, this kind of memory-efficiency can only be exploited when the same item is enqueued and dequeued several times. In our case however, lock-free queues are used in the task scheduler where enqueued items correspond to activities that are currently suspended. We assume that long-living activities are suspended and resumed frequently especially when there are more running activities than processors. By reusing intermediate nodes, this approach allows to save all implicit memory allocations caused by suspending and thereby enqueueing an activity.

Listing 3.6 on the facing page shows the extended data structure and queue operations implementing the idea of reusing intermediate nodes. All changes with respect to the previous version are labelled accordingly. In particular, the unconditional allocation of a new node at the beginning of the ENQUEUE procedure has been replaced by a reuse of the node associated with an item if available. Additionally, the deallocation of the previous sentinel node at the end of the DEQUEUE procedure has been substituted for a pairing of that node with the returned item. The assumption here is that the user will finally deallocate the node associated with a dequeued item as soon as the item is no longer used. The deallocation can also be automatically performed by a corresponding finaliser or deconstructor of the ITEM data structure in case that notion is supported by the programming language.

In both procedures, the modification of the associated node are in regions where the item is completely owned by the caller of the operation. Except for the pairing of the items with their intermediate nodes, nothing else has been altered with respect to the algorithm shown in Listings 3.4 and 3.5. In particular, both algorithms have the same set of linearisation points. Therefore, they seem to share the same correctness and liveness guarantees. However, reusing nodes is extremely prone to the ABA problem.

The ABA problem was first mentioned in the manual of the compare-and-swap operation of the IBM System/370 [IBM83] and occurs when one activity fails to recognise that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed. In our case, the ABA problem is triggered when a dequeue operation does not recognise that the queue has been modified in the meantime. Since the nodes are reused whenever possible, it could very well happen that the queue ends up having exactly the same sentinel node before and after the execution of several queue operations. If a concurrent activity expects the same sentinel node while dequeueing, the corresponding atomic compare-and-swap operation at linearisation point D_3 will succeed. The activity therefore falsely assumes that the contents of the queue have not been modified and effectively cancels all operations that happened in the meantime.

```
structure ITEM
    node : pointer to NODE                          ▷ node associated with item

procedure ENQUEUE(item, queue)
    node ← item→node                                ▷ reuse node associated with item
    if node = null then
        node ← ALLOCATE()
    node→next ← null
    node→item ← item
    repeat
        last ← CAS(queue.last, null, null)
        next ← CAS(last→next, null, node)           ▷ E1
        if next ≠ null then
            CAS(queue.last, last, next)             ▷ E2
    until next = null
    CAS(queue.last, last, node)                     ▷ E3

procedure DEQUEUE(queue)
    repeat
        first ← CAS(queue.first, null, null)
        next ← CAS(first→next, null, null)
        if next = null then
            return null                             ▷ D1
        CAS(queue.last, first, next)                ▷ D2
        item ← next→item                            ▷ store a local copy of the item
    until CAS(queue.first, first, next) = first     ▷ D3
    item→node ← first                               ▷ associate free node with item
    return item
```

Listing 3.6: Data structure and operations of a lock-free queue reusing intermediate nodes by pairing them with the actual items.

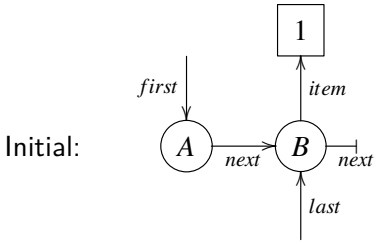

Figure 3.3: Example of a queue containing a single item.

The initial state of one of the simplest cases where the ABA problem occurs is depicted in Figure 3.3. It shows an example of a queue consisting of a linked list of two nodes A and B where A is the sentinel and B points to the single item of the queue. Figure 3.4 on the next page shows a sequence of linearisation points reached by two concurrent activities P and Q operating on this initial queue. The sequence reads as follows and triggers the ABA problem in the end:

1. Both activities P and Q try to dequeue the single item from the queue. Both reach D_2 but do not modify the reference to the last node since it is already up to date. For the illustration of the problem, we assume that only P proceeds with D_3 whereas Q does not execute it until step four below. However, Q still remembers node A as the first node and B as its successor.

2. Activity P now completely enqueues the same item again by reaching linearisation points E_1 and E_3. The enqueue operation reuses the node A as it was associated beforehand with the enqueued item.

3. Activity P immediately dequeues the item again by reaching $P|D_2$ followed by $P|D_3$ and leaves behind an empty queue consisting only of the sentinel node A. This time node B becomes associated with the dequeued item.

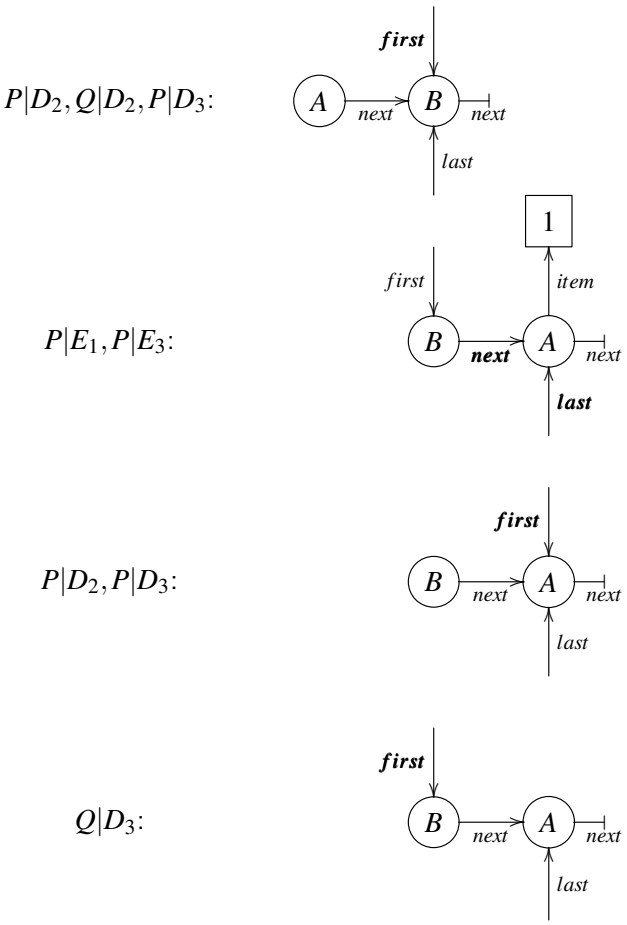

Figure 3.4: States of the same queue after two activities P and Q reach different linearisation points while performing several enqueue and dequeue operations showing the ABA problem.

4. Only now activity Q continues and reaches D_3. The corresponding compare-and-swap operation succeeds because the reference to the first node matches with the expected node A. This is a false positive since the queue is currently empty and any dequeue operation should return prematurely instead. Hence, activity Q erroneously advances the reference to the first node to B although the current sentinel A has no successor. The result of this occurrence of the ABA problem is a linked list that consists of two sentinel nodes and no actual item.

The very same problem can also occur when we use the algorithm from the previous section but discard its simplistic assumption that all allocated nodes occupy different memory regions. In this case, it is only a matter of time before the memory manager reuses the memory of a deallocated node again when allocating a new node. As a consequence, a new allocated node may have the same reference value as a previously deallocated one. In this case the node is reused as well but this time by the memory manager rather than by the algorithm itself. In the end however, the effect is the same and the ABA problem can occur just as easily.

In the sample sequence of linearisation points showed above we assumed that one activity performs several complete queue operations before the other one finishes a single one. This might seem unrealistic but is not entirely impossible since the slower activity might have been suspended by a preemptive scheduler. Using cooperative multitasking we can weaken the probability of this scenario by wrapping both procedures in uncooperative blocks which prevent voluntary task switches altogether. However, on an interrupt driven system an interrupt may happen at any time and cause additional delay in the execution of the queue operation. In addition, activities running on distinct processors may be executed at different speeds. In practice, the ABA problem does still occur and we have been able to reproduce at will.

In the standard literature, there exist several different approaches to cope with the ABA problem in general. All of them are more or less useful in practice but only few solve the problem conceptually. The four most common techniques are as follows:

1. One solution for most lock-free data structures is the application of the double compare-and-swap operation called DCAS. It allows to compare two unrelated memory locations instead of one in order to be sure that the overall state has changed indeed. This technique has for example been used by Massalin and Pu [MP91, Mas92] and Greenwald and Cheriton [GC96, Gre99]. Unfortunately, this atomic operation is not as well supported on common modern hardware architectures as its single compare-and-swap variant.

   Regarding our lock-free queue implementation, this operation could help to check simultaneously whether the local copy of the sentinel as well as its immediate successor are still the same. Another possibility is to introduce an additional variable in the queue data structure which counts the number of all previous modifications in order to keep track of all temporary changes.

2. Being limited to the single compare-and-swap operation supported by almost all multiprocessor hardware architectures, it has been tried several times to mimic the behaviour of DCAS. This solution is generally referred to as pointer tagging or pointer stamping and encodes the number of modifications inside the shared reference instead of using a separate entity. This technique has been employed multiple times by Michael and Scott [MS96, Mic04a].

   In practice, the allocated memory for representing an intermediate node intentionally gets aligned on a certain word boundary such that some of the least significant bits of pointers to this data structure can be repurposed for counting the modifications. However, if the ABA problems happens often enough in sequence, any set of bits will eventually overflow and pointer tagging is therefore not bullet-proof.

3. Another solution is trying to ensure that intermediate nodes are never reused as long as there are other activities that still reference them. This rule can be easily enforced by relying on a runtime environment that implements an automatic memory management in the form of a garbage collector [GPST09].

Regarding our lock-free queue, any node still referenced by an activity executing a dequeue operation is not reclaimed by the garbage collector and cannot be reused for the node allocation in the enqueue operation. However, this would imply to fall back on unconditionally allocating nodes as in the previous version in order to leave the task of reusing nodes to the automatic memory management instead.

4. The same strategy of ensuring that nodes are not referenced by concurrent activities was also pursued by Michael [Mic04b]. Although his approach also solves the ABA problem as above, his main concern was safe memory deallocation in runtime environments without automatic memory management. His research was motivated by the question of how to manually deallocate objects safely in lock-free algorithms when there are concurrent activities potentially accessing these objects at the same time.

The next section describes the final iteration of our lock-free queue implementation which solves the ABA problem by drawing from the ideas of Michael for safe memory deallocation in lock-free programs.

3.5 Safe Memory Deallocation

There exist numerous runtime environments which do not have automatic memory management facilities such as garbage collectors. In addition, automatic memory management usually incurs a runtime overhead which for example may not be practical because of real-time considerations. Its alternative of manually deallocating data structures requires special attention by programmers in order to prevent applications from leaking memory. Correct manual memory management is even more difficult if the dynamically allocated data structures are potentially shared across several concurrent activities. This especially applies to data structures used in lock-free algorithms since there is no proper synchronisation of concurrent accesses. The corresponding memory can only be deallocated safely if no other activity is accessing it any more.

The problem of safe memory deallocation has been addressed several times using techniques like epoch based reclamation [Fra04] or reference counting [GPST09]. Michael proposed the notion of hazard pointers which identify all objects that are about to be used by concurrent activities [Mic04b]. The basic idea is to announce to contending activities that memory is potentially still in use and must therefore not be deallocated. If this is the case, the reclamation of the memory has to be deferred until there are no other activities referencing it any more. As a consequence, memory is deallocated and potentially reused if and only if it is referenced by a single activity which nicely solves the ABA problem. If the memory is reclaimed too early on the other hand, any subsequent dereferencing of pointers to this memory region is unsafe and therefore called hazardous.

An interesting question regarding this approach is how to represent hazard pointers and make them available to all activities. Obviously, the property of whether an object is still referenced cannot be stored in that object itself since that would require an activity to access it in order to set or query that information. This implies a race condition because the corresponding memory could have already been deallocated in the meantime. Therefore, hazard pointers have to be stored and identified by their actual value.

The challenge of making hazard pointers usable in lock-free algorithms is to implement their lookup efficiently and their data structures in a lock-free manner. Michael proposed a data structure called hazard pointer record which stores the values of all hazard pointers of a single activity. The set of all hazard pointers in the system is represented by a linked list of all hazard pointer records which grows for each additionally participating activity. This list is accessible to all activities and must be traversed in order to check whether a pointer is actually hazardous or not. The number of required hazard pointers per activity at any given point in time depends on the actual non-blocking algorithm but is usually not higher than two [Mic04b].

Using the notion of processor-local storage as described in Section 2.3.3 we were able to simplify and optimise the representation of hazard pointers in general. We show how this technique is used by applying it to a final iteration of our lock-free queue implementation. The result is an unbounded and memory-efficient queue that does not suffer from the ABA problem.

```
structure PROCESSOR
    hp1 : pointer to NODE
    hp2 : pointer to NODE
    pool1 : pointer to NODE
    pool2 : pointer to NODE

variable
    cpu : array N of PROCESSOR
```

Listing 3.7: Generic data structure for storing hazard pointers for a system with N processors.

Listing 3.7 shows a generic data structure for storing hazard pointers for a system with N processors. It consists of a global array of records which stores a pair of hazard pointers and a pair of so-called pooled nodes for each processor in the system. A pooled node is a currently free node whose reuse has been deferred because it might still be hazardous.

The advantage of this representation is that the complete set of hazard pointers is stored in a single array which can be traversed efficiently for lookup. It can be extended easily for algorithms that need more than two hazard pointers and is therefore universally applicable in any lock-free data structure. Each activity can efficiently and concurrently modify its hazard pointers by using the unique index of the currently executing processing unit called *processor* hereafter. The precondition is that the corresponding section of the code which maintains hazard pointers is always executed by the same processor. With the help of cooperative multitasking, we can embed the corresponding code in an uncooperative block which guarantees that there are no switches to other tasks or processors during its execution.

The management of potentially reused nodes including the lookup of hazard pointers is shown in Listing 3.8 on the facing page. The corresponding algorithm for safely recycling a node is shown in the ACQUIRE procedure. It either returns a node that is not hazardous and therefore safely reusable, or an invalid reference if no such node could be found.

```
procedure SWAP(reference to local, reference to shared)
    uncooperative
        node ← CAS(shared, null, null)
        if CAS(shared, node, local) = node then
            local ← node

procedure ACQUIRE(node)
    uncooperative
        repeat
            for all c ∈ cpu do
                if node = null then
                    return null
                else if node = c.hp1 then
                    SWAP(node, c.pool1)
                else if node = c.hp2 then
                    SWAP(node, c.pool2)
        until no more swapping occurred
        return node
```

Listing 3.8: Wait-free node acquisition for safe reuse.

The ACQUIRE procedure takes a node as argument and checks whether it is hazardous by comparing its value against the complete set of hazard pointers. If there is a match, the node may not be reused safely since another activity has indicated that it is still potentially using it. In order to defer the reuse of the node, its reference is atomically exchanged with the pooled node of the corresponding hazard pointer using the SWAP procedure. The hazardous node becomes pooled and the algorithm continues with the previously pooled node. Because the resulting reference could still be hazardous, it has to be compared against all remaining hazard pointers. The algorithm continues checking as long as the node reference is valid and further exchanges were necessary. Hence, the returned node reference is either invalid or assuredly not hazardous.

Whenever the node is potentially hazardous, it is atomically exchanged with the pooled node that corresponds to one of the hazard pointers of a processor. Since lock-free algorithms are constrained to store distinct values in their hazard pointers, the set of pooled nodes therefore always contains pairwise different entries per processor. Because the node initially passed as argument is different from all pool entries, there is always at least one more node available than nodes referenced by hazard pointers. Thus, no more than $2 \cdot N$ exchanges are required until a node is found that is not referenced by any hazard pointer. As a result, the loop always terminates and renders the whole operation wait-free.

Listing 3.9 on the next page shows the ENQUEUE and DEQUEUE procedures of our final queue implementation. Both procedures are similar to the previous version and therefore use the same queue data structures as before. This time however, the application of processor-local storage to access the hazard pointers requires both operations to be executed in uncooperative blocks. All other changes are labelled accordingly and include the node acquisition at the beginning of the ENQUEUE procedure as well as code that sets and resets hazard pointers. All of these changes do not influence the correctness and liveness properties of the previous version.

As before, the assumption of this algorithm is that the user or a respective finaliser reclaims the node associated with an item whenever the latter is no longer used and is deallocated. In order to reclaim it safely, its reference can be passed to a call of the ACQUIRE procedure. The node returned by that procedure may potentially be another one but is guaranteed not used by any other activity. This allows to safely deallocate either the node associated with the item or a previously pooled one and no memory is leaked.

The shown technique of acquiring and accessing nodes is not bound to our particular lock-free queue implementation in any way. In fact, it is a very generic programming idiom and can be universally applied to all lock-free algorithms that employ intermediate nodes. The only difference may be a higher number of hazard pointers that are simultaneously required by an algorithm [Mic04b]. In this case, only the sets of hazard pointers and pooled nodes have to be increased in order to cover the differing setting while the underlying node acquisition can be reused as is.

```
procedure ENQUEUE(item, queue)
    uncooperative
        node ← ACQUIRE(item→node)                ▷ Safe node acquisition
        if node = null then
            node ← ALLOCATE()
        node→next ← null
        node→item ← item
        repeat
            last ← ACCESS1(queue.last)           ▷ Access shared node
            next ← CAS(last→next, null, node)
            if next ≠ null then
                CAS(queue.last, last, next)
        until next = null
        cpu[processor].hp1 ← null                ▷ Reset hazard pointer
        CAS(queue.last, last, node)

procedure DEQUEUE(queue)
    uncooperative
        repeat
            first ← ACCESS1(queue.first)         ▷ Access shared nodes
            next ← ACCESS2(first→next)
            if next = null then
                cpu[processor].hp1,2 ← null      ▷ Reset hazard pointers
                return null
            CAS(queue.last, first, next)
            item ← next→item
        until CAS(queue.first, first, next) = first
        cpu[processor].hp1,2 ← null              ▷ Reset hazard pointers
        item→node ← first
        return item
```

Listing 3.9: Lock-free queue operations with safe node reuse.

```
procedure ACCESS_index(reference to shared)
    uncooperative
        repeat
            node ← CAS(shared, null, null)
            cpu[processor].hp_index ← node
        until CAS(shared, null, null) = node
        return node
```

Listing 3.10: Generic operation to safely access a shared reference by copying its value into a processor-local hazard pointer.

Listing 3.10 finally shows the generic ACCESS procedure used to update the value of a single hazard pointer. An algorithm must call this operation before it can safely dereference a shared node. The procedure first copies the value of the shared reference given as argument into the processor-local hazard pointer identified by the indices *processor* for the current processing unit and *index* for the respective instance. It then compares the copy again with the value of the shared reference which might have already changed and repeats the update of the hazard pointer if necessary. This check is required to ensure that contending activities have seen the new value of the hazard pointer before erroneously assuming that a node is not hazardous. The value returned by this procedure is the current value of the shared reference given as argument. The whole operation does not obstruct other activities since it only modifies processor-local data. It is therefore lock-free and safe from concurrent access.

Concerning an activity that fails to make progress in a lock-free algorithm for whatever reason, it is possible that a hazard pointer is never invalidated any more. As a consequence, all contending activities naturally assume that the corresponding node is still in use. The hazardous node therefore gets pooled and can never be reused again causing a new node to be allocated instead. However, the set of pooled nodes is limited by the number of hazard pointers and is thus never higher than $2 \cdot N$ at any given time. Therefore, the amount of surplus allocations is bounded by the same number.

By combining lock-free programming with cooperative multitasking, we finally reached our goal of developing a practical, safe, and memory-efficient implementation of an unbounded and lock-free queue. This work forms the foundation of the cooperative scheduler described in the next chapter where queues are used extensively for lock-free multitasking.

4 Lock-Free Scheduling

This chapter describes all components of the cooperative task scheduler used in our runtime system. It first explains the fundamental lock-free building blocks which form a framework for implementing more sophisticated synchronisation primitives.

4.1 Introduction

The term multitasking describes a technique that allows several independent tasks to progress concurrently by sharing the same execution environment. This shared environment includes common processing resources like a processor or main memory. A task switch is the process of divesting the currently running task of these resources and reassigning them to a different task. Multitasking therefore does not imply parallelism per se but it can convey that impression if task switches happen frequently enough. In general, the following two different types of task switches are distinguished:

- Synchronous task switches are initiated by a currently running task whenever it has to wait for some event to occur. It voluntarily gives up the control of execution and explicitly leaves it to some other task that is ready to run. The kind of event it waits for can be the occurrence of an interrupt or a condition that has to be met by another task.

- Asynchronous task switches are initiated by external events like periodical interrupts and are in general not under the control of the currently running task. This kind of task switch is typically completely transparent to the currently running task since it is preempted at an arbitrary location during its execution.

The process of determining the next running task and the duration of its execution until the next task switch is generally called scheduling. A preemptive scheduler employs asynchronous as well as synchronous task switches in order to implement multitasking. Asynchronous task switches offer the advantage that a program does not have to be prepared for the execution in a multitasking environment. Since the preemption is completely transparent to the program, a preemptive scheduler allows to execute arbitrary programs concurrently. This is why preemptive scheduling is a popular choice for modern operating systems running on machines that allow to dynamically load arbitrary programs interactively.

However, asynchronous task switches are generally considered expensive since they preempt programs at arbitrary locations during their execution. As a consequence, the subset of the common processing resources that have to be stored and restored upon consecutive task switches can usually not be known in advance. In order to be prepared for the worst case, the complete state of the processor including the values of the program counter, the stack and frame pointers, and all potentially modified processor registers has to be recorded when it is reassigned to a different task.

Table 4.1 on the facing page for example shows the processor registers that have to be potentially restored on a task switch performed on the AMD64 architecture [Adv13]. The corresponding processor context takes up a total of 736 bytes which has to be stored once for the suspended task and restored once for the resumed task during a task switch. Most of the 42 registers cannot be accessed in bulk and require the execution of separate instructions to get and set their values individually. This is often considered computationally expensive and implies an overhead since most of the time only a small subset of the execution environment is actually in use.

In order to compensate for the runtime costs of asynchronous task switches, preemptive schedulers often try to delay them as long as possible. The corresponding time window in which tasks are allowed to execute without being preempted is usually called a quantum or a time slice. The schedulers of modern operating systems like Windows and Linux based systems commonly choose a quantum with a duration between ten and few hundred milliseconds [RSI12, Lov10].

Application Registers	Count	Size (Bytes)
64-bit general-purpose registers	16	128
80-bit floating-point registers	8	80
256-bit vector computing registers	16	512
64-bit program counter	1	8
64-bit program flags	1	8
Total	42	736

Table 4.1: Contents and memory consumption of the processor context on the AMD64 architecture [Adv13].

Considering a fair scheduler where each task is resumed on a regular basis, the length of the quantum directly affects the overall idle time of a task as well as its observable responsiveness under high load. As a result, there is a trade-off between optimising the costs of asynchronous task switches and maintaining interactivity when there are a lot of task switches to be scheduled.

Regarding synchronous task switches on the other hand, tasks always pause themselves at predetermined locations. Since they give up the control of execution explicitly, the set of processing resources that has to be restored after the synchronous task switch is known by the task itself. Explicit task switches are most often realised by calling corresponding functions offered by the scheduler or runtime system. In this case, the processor state that has to be restored after the task switch is most often already covered by the underlying calling convention implemented by the compiler. In the simplest case, the compiler temporarily stores the required registers on the stack when calling a function and the remaining context information consists only of the program counter and the stack pointers. As a result, synchronous task switches are typically much more efficient than asynchronous task switches because their runtime cost is equivalent to ordinary function calls. There is hardly any trade-off that has to be considered and significantly smaller time slices can be used if necessary.

A further advantage of synchronous task switches is the fact that they can be implemented exclusively in software since they do not rely on external events like hardware interrupts. The implementation of synchronous task switches can therefore be written in a completely portable manner, whereas asynchronous task switches always require some sort of machine-dependent interrupt handling and processor state representation.

In order to achieve a highly portable runtime system, we abandoned the idea of preemptive scheduling which requires asynchronous and therefore machine-dependent task switches. Instead, our scheduler implements only synchronous task switches which are completely portable. The result is a cooperative scheduler which relies on tasks relinquishing control voluntarily from time to time in order to cooperate with all other tasks in the system.

4.2 Foundations

In this section we present a lock-free scheduler for cooperative multitasking of lightweight activities on multiple processors. As explained above, since it is based only on synchronous task switches, it does not rely on hardware specific features and can be implemented completely in software. Gidenstam and Papatriantafilou, authors of the LFTHREADS library, were amongst the earliest to give a proof of concept of this approach [GP07].

The basic foundation of the scheduler is a simple task descriptor called ACTIVITY shown in Listing 4.1 on the next page, which represents an activity associated with an active object. An activity is an extension of the ITEM data structure and can therefore be enqueued in and dequeued from instances of the lock-free queue presented in Section 3.5. The descriptor stores information about its stack, the associated active object, as well as specific properties like its current priority, the index of the processor it is running on and the remaining length of its quantum.

In contrast to the A2 operating system where processes can be in one of several different states [Mul02], activities in our runtime system have only two implicit states. They are either currently running and executed on a processor with the given index, or they are currently suspended in which

```
structure ACTIVITY extends ITEM
    object : object                          ▷ Associated active object
    quantum : integer                        ▷ Time quantum
    priority : integer                       ▷ Task priority
    index : integer                          ▷ Processor index
    stack : pointer to array size of byte    ▷ Stack representation
    limit : address                          ▷ Lowest valid stack address
    frame : address                          ▷ Frame pointer
    finalizer : procedure                    ▷ Task switch finaliser
    argument : any                           ▷ Argument for finaliser
    previous : pointer to ACTIVITY           ▷ Suspended activity
```

Listing 4.1: Representation of an activity associated with an active object.

case their descriptor is or is about to be enqueued in a lock-free queue. This generic simplification still covers all process states of the original A2 system but allows at the same time to express the transition between the two remaining states in a lock-free manner. In addition, the lack of an explicit process state allows the creation of arbitrary synchronisation primitives on top of the scheduler.

Each activity has its own stack memory represented by an array of bytes that embodies its complete call stack. Stacks correspond to ordinary memory blocks on the heap and can therefore have an arbitrary size. They are designed to grow dynamically in order to accommodate activities which require larger call stacks, see Section 5.2. The total number of activities is therefore only limited by the available memory in the system.

The stack also stores the context of an activity during a task switch. In general, this context consists of the stack represented by stack and frame pointers as well as the values of application-specific hardware registers. These registers are handled by the compiler, when it has to follow a calling convention which requires them to be temporarily pushed on the stack during function calls. Since all synchronous task switches are initiated by

ordinary function calls, activities need not represent any processor state besides the stack, because all of the context which is required to be restored afterwards is already stored by the compiler. The same applies to the program counter of an activity which is also pushed on the stack because of the function call. Therefore, a synchronous task switch only has to exchange the stack and frame pointers just before returning from the corresponding function call.

The actual algorithm of task switching is shown in Listing 4.2 on the facing page. The SWITCHTO procedure first gets access to the descriptor of the currently running activity in order to store its current context. The procedure makes use of two functions called GETACTIVITY and GETFRAMEPOINTER which are provided by the compiler and allow to query the values of the registers storing the current activity and the stack frame pointer in a portable manner. Since the actual stack pointer and the program counter of the function caller are already pushed on the stack by the function prologue generated by the compiler, it suffices to store the address of the current stack activation frame.

In a second step, the procedure prepares the given activity for the task switch by resetting its quantum to a default value and also supplying the index of the currently executing processor. The actual task switch is performed in the last step, where the context of the new activity is restored using the procedures SETACTIVITY and SETFRAMEPOINTER provided by the compiler. The stack pointer and the program counter are finally restored by returning from the procedure which pops the corresponding values from the stack automatically.

4.2.1 Task Switch Finalisers

By returning to a different activity, the task switch changes the implicit state of the previously running activity to suspended, while the new activity changes from suspended to currently running. However, there is a subtlety which prevents the suspended activity from being placed in a queue. The reason is that the SWITCHTO procedure is technically speaking executed in its entirety by the same activity.

```
procedure SWITCHTO(activity, finalizer, argument)
    uncooperative
        current ← GETACTIVITY()                          ▷ Store context
        current→frame ← GETFRAMEPOINTER()
        activity→quantum ← Default                       ▷ Prepare activity
        activity→index ← current→index
        activity→finalizer ← finalizer
        activity→previous ← current
        activity→argument ← argument
        SETACTIVITY(activity)                            ▷ Restore context
        SETFRAMEPOINTER(activity→frame)
        return
```

Listing 4.2: Basic context switching for activities.

Although the return operation pops the new value of the stack pointer from the stack frame of the resumed activity, the stack pointer used within the SWITCHTO procedure still refers to the stack of the calling activity. The actual context switch is therefore not completed until the return instruction has been executed. If the calling activity would have been enqueued in a lock-free queue before the task switch is completed, another processor could dequeue that activity immediately. This is a severe problem if that processor switches to the dequeued activity before the original processor completed the first task switch. In this disastrous case, the same activity is actually running on two different processors which share the same stack pointers and therefore probably corrupt the stack.

Consequently, the enqueue operation must always be executed after the SWITCHTO procedure has returned to the resumed activity. A basic solution to this problem would be the execution of the actual task switch including the enqueue operation by a third designated activity created solely for this purpose. A much more generic solution is provided by the notion of task switch finalisers which we introduced in order to execute arbitrary code on behalf of the suspended activity after a task switch has completed.

```
procedure FINALIZESWITCH( )
    uncooperative
        current ← GETACTIVITY()
        current→finalizer(current→previous, current→argument)
        return
```

Listing 4.3: Finalising task switches on behalf of the suspended activity.

Task switch finalisers are basically function pointers given as an argument to the SWITCHTO procedure which supplies it to the resumed activity. A call to this procedure must always be followed immediately by a call to the FINALIZESWITCH procedure which in turn executes the supplied task switch finaliser as shown in Listing 4.3. This convention allows to execute an arbitrary procedure in the context of the resumed activity before it continues its own execution. The concept of executing code on behalf of other activities is indispensable for implementing synchronisation primitives in a lock-free manner. Unconditionally placing the suspended activity in a lock-free queue is just one of many applications.

4.2.2 Basic Scheduling

Besides providing support for basic task switching, the scheduler also keeps track of all activities that are currently suspended but otherwise ready to be resumed by any processor. Ready activities are placed in global lock-free queues called ready queues. The first-in first out character of this data structure ensures that activities are scheduled in a round robin fashion.

The scheduler currently supports a simplistic priority model and maintains a separate global ready queue for each of the four increasing priorities nominally called idle, normal, high, and real-time. The corresponding data structure and the procedure for selecting the next activity to be switched to is given in Listing 4.4 on the next page. The SELECT procedure tries to dequeue a single activity from the ready queue associated with the highest priority and continues with the next lower priority until an activity is found or the minimal priority given as argument has been reached.

```
variable
    readyQueue : array 4 of QUEUE

procedure SELECT(minimalPriority)
    uncooperative
        for all priority ∈ {Realtime, High, Normal, Idle} do
            if priority >= minimalPriority then
                activity ← DEQUEUE(readyQueue[priority])
                if activity ≠ null then
                    return activity
        return null
```

Listing 4.4: Selecting activities from ready queues with different priorities.

```
variable
    working : integer

procedure RESUME(activity)
    uncooperative
        ENQUEUE(activity, readyQueue[activity→priority])
        if CAS(working, 0, 0) < Processors then
            RESUMEANYPROCESSOR()
```

Listing 4.5: Entering suspended activities into ready queues.

A suspended activity can be entered into a ready queue by calling the RESUME procedure shown in Listing 4.5. As soon as the activity is enqueued by this procedure, it is available for selection by the same or another processor and a task switch to this activity can be performed. The RESUME procedure additionally keeps track of an atomic counter called *working* which indicates the number of processors that are currently not idle. If there is an idle processor, it is resumed by a call to the RESUMEANYPROCESSOR procedure which is described in more detail in Section 4.4.

```
procedure SWITCH( )
    uncooperative
        current ← GETACTIVITY()
        activity ← SELECT(current→priority)
        if activity ≠ null then
            SWITCHTO(activity, ENQUEUESWITCH, current→priority)
            FINALIZESWITCH()
        else
            current→quantum ← Default

procedure ENQUEUESWITCH(previous, priority)
    uncooperative
        ENQUEUE(previous, readyQueue[priority])
        if priority ≠ Idle then
            if CAS(working, 0, 0) < Processors then
                RESUMEANYPROCESSOR()
```

Listing 4.6: Performing cooperative task switches.

4.2.3 Cooperative Task Switches

Cooperative multitasking relies on each activity to relinquish the control of execution voluntarily to another activity on a regular basis. In order for activities to cooperate with each other, the scheduler provides a global procedure called SWITCH which is shown in Listing 4.6. It selects a suspended activity that has to be resumed and suspends the calling activity by enqueueing it into the corresponding ready queue. The actual enqueue operation is performed by a task switch finaliser called ENQUEUESWITCH which is executed by the resumed activity. This ensures that the suspended activity is not available for resuming by another processor until the task switch is actually completed. If there is no activity with the same or higher priority available, there is no task switch necessary and the activity can continue with a default quantum.

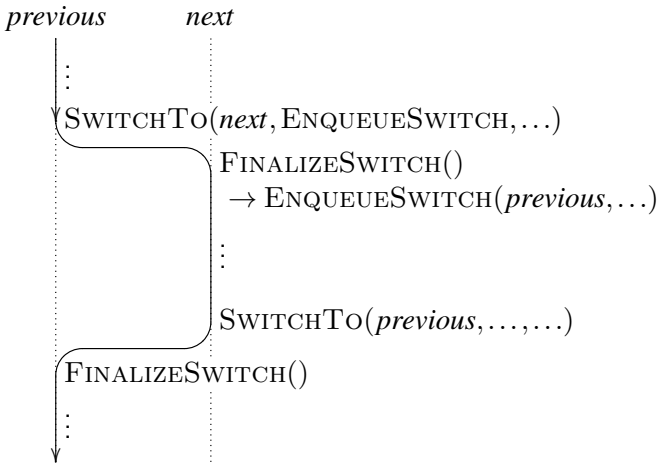

Figure 4.1: Sample control flow of a single processor executing two consecutive task switches between two activities.

A concrete example of the control flow on a single processor executing two consecutive task switches is depicted in Figure 4.1. This sample scenario involves two activities called *previous* and *next* which pass the control of execution to each other. After its time quantum has expired, activity *previous* calls the SWITCH procedure which in turn executes the SWITCHTO procedure in order to initiate a task switch to *next* and passes ENQUEUESWITCH as task switch finaliser. Before activity *next* resumes its previously suspended execution, it first calls the FINALIZESWITCH procedure which indirectly calls the task switch finaliser of *previous*. After a while, *next* eventually relinquishes the control of execution and passes it again on to *previous* by calling the SWITCHTO procedure correspondingly. By returning from its previous call of the SWITCHTO procedure, activity *previous* continues its execution where it left off before but first finalises the task switch of *next*.

Each call to the SWITCH procedure in a program represents an additional entry point for suspending and resuming its execution. Any subroutine calling this procedure is therefore automatically turned into a coroutine according to Conway and Knuth [Con63]. However, if a program fails to call this procedure during its execution for whatever reason, there will be no task switches on the corresponding processor. This uncooperative behaviour can have a severe impact on the reliability, responsiveness, and correctness of the whole runtime system. The next section shows how to cope with this problem and to ensure that programs are always cooperative.

4.2.4 Implicit Cooperative Multitasking

Usually, systems with cooperative multitasking rely heavily on the programmer to insert cooperative task switches at appropriate locations in the code. It is hard to prove that arbitrary programs are indeed cooperative in this respect even if their source code is available for inspection. This might have been one additional reason why preemptive multitasking is almost ubiquitously applied in modern multiprocessor operating systems, since preemption does not depend on this kind of correctness.

Another approach to guarantee the cooperativeness of arbitrary programs is to let the compiler automatically insert task switches into the translated machine code where necessary. We call this implicit cooperative multitasking since a program compiled with this development tool does not require the programmer to insert any explicit task switches in order to be correct with respect to cooperativeness. As a result, source code compiled with enabled implicit cooperative multitasking looks exactly the same as code that is executed with a preemptive scheduler in mind. It is therefore irrelevant for programmers which kind of scheduler their program is targeted at, as long it is recompiled accordingly.

The actual code inserted by the compiler checks whether the quantum of the current activity has expired and calls the SWITCH procedure if necessary. For this purpose, the compiler knows the layout of the ACTIVITY data structure and reserves a dedicated general-purpose register that stores the descriptor of the currently running activity.

```
    sub   [rcx + 88], 10  ; decrement quantum by 10
    jge   skip            ; check if it is negative
    call  Switch          ; perform task switch
skip:
```

Listing 4.7: Inserted instruction sequence for implicit task switches on the AMD64 architecture.

In order to stay portable for the implementation of the runtime system and keep the check as small as possible, the compiler does not measure the time between two consecutive checks but rather the amount of generated instructions. The actual duration of hardware operations usually varies amongst different instructions and is obviously machine-dependent, but counting the number of instructions has the advantage that the result is always constant and statically known while translating the code. The quantum is therefore not related to actual execution time but rather stores the number of instructions an activity is allowed to execute until the next cooperative task switch must occur. However, task switches are always synchronous which is why the quantum can be chosen to be rather small and does not need to be time specific.

Listing 4.7 shows an example of an implicitly generated instruction sequence of a task switch check after ten instructions targeting the AMD64 hardware architecture [Adv13]. Here, the dedicated general-purpose register pointing to the descriptor of the current activity is called `rcx` while its quantum has an offset of 88 bytes into the ACTIVITY structure. The check consists of only three instructions requiring twelve bytes in total.

The compiler inserts this sequence whenever a program executes a loop statement or calls a recursive procedure. From a low-level perspective, the compiler inserts the checks for loops whenever it emits backward branches in the code. This generic approach basically covers all possible looping control-flow constructs of high-level programming languages. Checks are also inserted after ordinary statements when the number of instructions that have been generated since the last check exceeds a predefined limit.

This basic policy guarantees that the activity always behaves cooperatively even if it executes infinite loops or otherwise does not make observable progress. A similar technique of code instrumentation for software-based multitasking has also been proposed by Bläser [Blä07]. His dedicated general-purpose register directly represents the quantum of the currently executing activity and its value is decremented by periodical timer interrupts. This approach allows to express the quantum in time specific units but is obviously not as portable since it requires the runtime system to handle timer interrupts accordingly.

As specified in Section 2.3.1, uncooperative blocks prevent the compiler from inserting any checks inside a statement block. All procedures shown so far in this chapter are directly or indirectly called by an implicit task switch generated by the compiler. This is why all of them have been marked as uncooperative in order to prevent any potential infinite recursion triggered by an implicit task switch.

4.3 Synchronisation Primitives

The previous section established a solid foundation of basic scheduler operations. This section shows how this generic framework can be used to build sophisticated synchronisation primitives in a lock-free manner. It exemplifies three popular constructs which build upon each other including a fully-fledged monitor for object oriented programming languages such as Active Oberon.

4.3.1 Events

A simple primitive for synchronising activities is an event, also known as an event semaphore, which allows arbitrary activities to wait for an event to occur [HS08]. A simplistic implementation of an event is given in Listing 4.8 on the facing page. The corresponding data structure consists simply of a lock-free queue which stores all activities waiting for the event to be signalled.

```
structure EVENT
    waiting : QUEUE

procedure WAIT(event)
    uncooperative
        repeat
            activity ← SELECT(Idle)
        until activity ≠ null
        SWITCHTO(activity, ENQUEUEWAIT, event)
        FINALIZESWITCH()

procedure ENQUEUEWAIT(previous, event)
    uncooperative
        ENQUEUE(previous, event.waiting)

procedure SIGNAL(event)
    uncooperative
        repeat
            activity ← DEQUEUE(event.waiting)
            if activity ≠ null then
                RESUME(activity)
        until activity = null
```

Listing 4.8: Data structure and operations of a simplistic event synchronisation primitive.

The waiting queue is filled with activities calling the WAIT procedure. This procedure first repeatedly tries to select any activity that is ready to run and then performs a task switch to that activity. The corresponding task switch finaliser called ENQUEUEWAIT is responsible for placing the suspended activity into the waiting queue of the event.

The selection of a ready activity can fail if all other processors are concurrently performing a cooperative task switch but their task switch finalisers have not yet placed the suspended activities into the ready queue. The chances that all ready queues are empty is low since all processors are making progress towards the completion of their task switch finalisers. The failed selection is therefore harmless and can just be repeated yielding a ready activity with a high probability. This kind of race condition however is a direct consequence of relying only on non-blocking scheduler operations.

The SIGNAL procedure finally dequeues all activities from the waiting queue of an event and resumes all of them. The simplistic assumption of this naive implementation is that no activity begins to wait while another activity executes this procedure. Otherwise, a resumed activity that waits immediately again on the same event could be resumed twice by the same signal operation. The next synchronisation primitive shows how to deal with this kind of problem.

4.3.2 Mutexes

A popular synchronisation primitive known as mutex or binary semaphore facilitates mutual exclusion [HS08]. A mutex can be acquired by a single activity in which case it is said to be owned by that activity. At the end of a critical section, the owner releases the mutex allowing it to be acquired again. A mutex is similar to an event, as it also must wake up activities that are blocked because of a failed acquisition and waiting for the mutex to be released.

Listing 4.9 on the next page shows the data structure and the acquire operation of a sample mutex implementation. A mutex consists of a queue of blocked activities and a reference to the current owner. The following actions are performed during the execution of the acquire operation:

```
structure MUTEX
    blocked : QUEUE
    owner : pointer to ACTIVITY

procedure ACQUIRE(mutex)
    uncooperative
        current ← GETACTIVITY()
        while CAS(mutex.owner, null, current) ≠ null do          ▷ (1)
            activity ← SELECT(Idle)                              ▷ (2)
            if activity ≠ null then
                SWITCHTO(activity, ENQUEUEAQUIRE, mutex)
                FINALIZESWITCH()

procedure ENQUEUEACQUIRE(previous, mutex)
    uncooperative
        ENQUEUE(previous, mutex.blocked)                         ▷ (3)
        if CAS(mutex.owner, null, null) = null then              ▷ (4)
            activity ← DEQUEUE(mutex.blocked)
            if activity ≠ null then
                RESUME(activity)
```

Listing 4.9: Data structure and acquire operation of a mutex synchronisation primitive.

1. The ACQUIRE procedure first tries to set the ownership of the mutex to the currently running activity using an atomic compare-and-swap operation. If the result is an invalid reference, the owner was set successfully and the mutex is now acquired. Otherwise, the mutex has already an owner and the current activity must be blocked.

2. The procedure selects any ready activity and performs a task switch using the ENQUEUEACQUIRE procedure as task switch finaliser. If there is no ready activity, the whole operation is repeated.

3. The EnqueueAcquire procedure is executed by the selected activity and places the suspended activity into the blocked queue of the mutex. Since there is no proper synchronisation, a concurrent release of the mutex could occur before, while, and after the enqueue operation. The ownership must therefore be rechecked in order to prevent the enqueued activity from remaining suspended forever. This check must occur after the enqueue operation because the check as well as the enqueue operation can be seen as linearisation points, and one of them has to precede the other one.

4. If the mutex is still acquired after the enqueue operation, it means that its owner or a follower will eventually release the mutex and dequeue the suspended activity. The same applies if the mutex was already released and immediately acquired again in the meantime.

   If the mutex is not acquired, a release operation must have been executed concurrently. If the release operation has already finished, the enqueue operation must be undone in order to prevent a lost wake-up call. Since queue data structures do not provide a corresponding operation, a dequeue operation must be performed instead. This yields either the suspended or any another blocked activity. In either case, resuming the activity probably allows it to acquire the mutex and a subsequent release operation will finally resume any blocked activity.

   Resuming too many activities is harmless because only one of them can acquire the mutex and all others will be blocked again. If the blocked queue was empty on the other hand, the suspended activity has already been dequeued and resumed.

The task switch finaliser used in the acquire operation is the first one that does not just place the suspended activity into some queue. It marks the first example where it is highly critical to recheck the condition which caused the previous activity to get suspended in the first place. This is necessary because there is no proper synchronisation between the evaluation of a condition and the corresponding enqueue operation.

```
procedure RELEASE(mutex)
    uncooperative
        current ← GETACTIVITY()
        CAS(mutex.owner, current, null)                    ▷ (1)
        activity ← DEQUEUE(mutex.blocked)                  ▷ (2)
        if activity ≠ null then
            RESUME(activity)
```

Listing 4.10: Release operation of the mutex synchronisation primitive.

In general, conditions for suspending an activity must always be evaluated again after the enqueue operation in order to cancel it if necessary. This interleaving of condition checking is a lock-free programming idiom that is used throughout the implementation of all synchronisation primitives.

The remaining release operation is shown in Listing 4.10. It is much simpler and performs the following steps:

1. The ownership of the mutex is invalidated using an atomic compare-and-swap operation which can only succeed if the RELEASE procedure was actually called by the current owner of the mutex.

2. One single blocked activity is resumed if there is one in the corresponding queue. This step cannot precede the previous one because the resumed activity could suspend itself immediately again and as a result remain suspended forever.

From this discussion we informally conclude that this lock-free implementation of a mutex is correct and does not suffer from the problems of the event synchronisation primitive shown before. However, this particular mutex implementation does not allow the mutex to be acquired recursively by the same owner. Moreover, releasing the mutex is not a fair action because resumed activities are only enqueued to the ready queue whereas activities acquiring the mutex for the first time receive preferential treatment. The synchronisation primitive discussed in the next section solves this problem by atomically transferring the ownership to the resumed activity.

4.3.3 Monitors

The final synchronisation primitive presented in this section is an example of a monitor for object-oriented programming languages. In the case of Active Oberon, a monitor is entered by executing a statement block marked as exclusive. This ensures mutual exclusion and allows several activities to synchronise with each other by awaiting conditions to be met. Monitors therefore combine the concept of mutexes and events.

Listing 4.11 on the facing page shows the data structure and the enter operation of our monitor implementation. Since we wish to allow nested acquisitions of a monitor, the data structure not only keeps track of the current owner, but also a counter called *level* which indicates the number of acquisitions by that owner. In addition, there are two distinct queues which contain activities that are either blocked because they failed to acquire the monitor or waiting for a condition to be satisfied.

The ENTER procedure is similar to the ACQUIRE procedure of the mutex and requires two parameters. The first one is the monitor to be acquired and the second one is the number of nested acquisitions to be performed by a single call of this procedure. The procedure performs the following actions:

1. Like the mutex acquisition in the previous example, the ownership of the monitor is set to the currently executing activity using a compare-and-swap operation. If this operation succeeds or the monitor is already owned by the calling activity, the procedure just increments the counter for nested acquisitions and returns.

2. As before, the currently running activity has to be suspended and enqueued into the blocked queue of the monitor if the acquisition fails. For this reason, the procedure selects any ready activity and performs a task switch using the ENQUEUEENTER procedure as task switch finaliser. If there is no activity ready to run, the procedure tries to acquire the monitor again without being suspended.

3. The ENQUEUEENTER procedure places the suspended activity into the blocked queue since that activity failed to acquire the monitor.

```
structure MONITOR
    level : integer
    blocked, waiting : QUEUE
    owner, sentinel : pointer to ACTIVITY

procedure ENTER(monitor, level)
    uncooperative
        current ← GETACTIVITY()
        loop
            previous ← CAS(monitor.owner, null, current)                ▷ (1)
            if previous = null ∨ previous = current then
                monitor.level ← monitor.level + level
                return
            activity ← SELECT(Idle)                                     ▷ (2)
            if activity ≠ null then
                SWITCHTO(activity, ENQUEUEENTER, monitor)
                FINALIZESWITCH()

procedure ENQUEUEENTER(previous, monitor)
    uncooperative
        ENQUEUE(previous, monitor.blocked)                              ▷ (3)
        if CAS(monitor.owner, null, null) = null then                   ▷ (4)
            activity ← DEQUEUE(monitor.blocked)
            if activity ≠ null then
                RESUME(activity)
```

Listing 4.11: Data structure and enter operation of a monitor synchronisation primitive.

4. As discussed above, the monitor could have already been released at any point in time between selecting an activity, switching to it, and enqueueing the suspended activity in the task switch finaliser. In this case, the blocked activity should not remain in the queue in order to prevent that it is never resumed again.

   Since the queue data structure does not support undoing the previous enqueue operation, the procedure performs a single dequeue operation instead. The resulting activity was the oldest one in the blocked queue and gets resumed. As before, this activity will in turn either successfully acquire the monitor or suspend itself again. In either case, the initially suspended activity will eventually be resumed.

The reverse operation of releasing the monitor is called EXIT and is shown in Listing 4.12 on the next page. It also takes two parameters indicating the monitor and the number of nested releases. The procedure performs the following actions:

1. The number of nested acquisitions is decremented. If this number does not reach zero, the monitor is still acquired and the procedure returns.

2. A single activity to be resumed is dequeued either from the waiting queue, or the blocked queue if there was no waiting activity. This order ensures that activities waiting for a condition to be satisfied take priority with respect to activities that are blocked while trying to acquire the monitor. This design known as eggshell model guarantees that any waiting activity is resumed as soon as another activity has satisfied its condition and released the monitor [ISO95]. This signalling regime has been proven by Dahl to be superior in terms of sequencing control and efficiency in comparison to its alternatives provided by modern programming languages like Java and C# [Dah99].

3. The ownership of the monitor is transferred to the dequeued activity using an atomic compare-and-swap operation. If there is no activity to be resumed, the owner is automatically released instead.

```
procedure EXIT(monitor, level)
    uncooperative
        monitor.level ← monitor.level − level                    ▷ (1)
        if monitor.level > 0 then
            return
        current ← GETACTIVITY()
        activity ← DEQUEUE(monitor.waiting)                      ▷ (2)
        if activity = null then
            activity ← DEQUEUE(monitor.blocked)
        CAS(monitor.owner, current, activity)                    ▷ (3)
        if activity = null then                                  ▷ (4)
            activity ← DEQUEUE(monitor.blocked)
        if activity ≠ null then
            RESUME(activity)
```

Listing 4.12: Exit operation of the monitor synchronisation primitive.

4. If the ownership was transferred in the previous step, the new owner is resumed. However, if the ownership was released instead of transferred, any activity that suspended itself beforehand will eventually be placed in the blocked queue of the monitor. If the corresponding task switch finaliser is finished before the ownership is released, the blocked activity could remain suspended forever and must therefore be dequeued and resumed.

Finally, monitors also allow activities that have acquired the monitor to wait for specific conditions to be met by contending activities. While waiting for a condition, the ownership is transferred to any other waiting or blocked activity in order to give it the chance to satisfy the condition. Since there might be several activities waiting for differing conditions, all of them have to be checked in succession. In order to prevent an endless loop, the monitor data structure additional contains a sentinel which indicates the first waiting activity in this sequence.

The corresponding AWAIT procedure is shown in Listing 4.13 on the facing page. This procedure is called by the Active Oberon compiler for each occurrence of an await statement. The compiler generates code that repeatedly checks the await condition and calls this procedure if it is not satisfied. The procedure performs the following actions:

1. Because the ownership of the monitor is released while waiting, the value of the corresponding nesting level is cached and set to zero.

2. If there is another waiting activity, the ownership is transferred to that activity by using the TRANSFERAWAIT task switch finaliser shown in Listing 4.14 on page 92. In order to prevent two or more waiting activities to transfer the ownership to each other in an infinite loop, the first activity waiting for a condition also sets the sentinel to refer to itself.

3. If the next waiting activity equals to the sentinel, all waiting activities checked their conditions but none could progress. The monitor must therefore be released in order to give blocked or newly arriving activities a chance to satisfy any of the unmet await conditions. The sentinel is reset and enqueued again in the waiting queue so that all waiting activities will recheck their conditions afterwards.

4. Since there is no waiting activity that can proceed at this point, any blocked activity is resumed instead. If there is a blocked activity, the ownership gets atomically transferred to it by performing a task switch using the TRANSFERAWAIT task switch finaliser shown in Listing 4.14. This procedure enqueues the suspended activity to the waiting queue of the monitor and atomically swaps the ownership.

5. If there was no blocked activity, any other activity ready to run is selected and gets the control of execution. The corresponding task switch finaliser called RELEASEAWAIT shown in Listing 4.14 enqueues the suspended activity into the waiting queue and invalidates the ownership. As before, any blocked activity must be resumed afterwards in order to prevent lost wake-up calls.

```
procedure AWAIT(monitor)
    uncooperative
        level ← monitor.level                                          ▷ (1)
        monitor.level ← 0
        current ← GETACTIVITY()
        activity ← DEQUEUE(monitor.waiting)                            ▷ (2)
        if activity ≠ null ∧ activity ≠ monitor.sentinel then
            if monitor.sentinel = null then
                monitor.sentinel ← current
            SWITCHTO(activity, TRANSFERAWAIT, monitor)
            FINALIZESWITCH()
        else
            monitor.sentinel ← null                                    ▷ (3)
            if activity ≠ null then
                ENQUEUE(activity, monitor.waiting)
            activity ← DEQUEUE(monitor.blocked)                        ▷ (4)
            if activity ≠ null then
                SWITCHTO(activity, TRANSFERAWAIT, monitor)
                FINALIZESWITCH()
            else
                activity ← SELECT(Idle)                                ▷ (5)
                if activity ≠ null then
                    SWITCHTO(activity, RELEASEAWAIT, monitor)
                    FINALIZESWITCH()
                else
                    CAS(monitor.owner, current, null)                  ▷ (6)
                    SWITCH()
        ENTER(monitor, level)                                          ▷ (7)
```

Listing 4.13: Await operation of the monitor synchronisation primitive.

```
procedure TRANSFERAWAIT(previous, monitor)
    uncooperative
        current ← GETACTIVITY()
        ENQUEUE(previous, monitor.waiting)
        CAS(monitor.owner, previous, current)

procedure RELEASEAWAIT(previous, monitor)
    uncooperative
        ENQUEUE(previous, monitor.waiting)
        CAS(monitor.owner, previous, null)
        activity ← DEQUEUE(monitor.blocked)
        if activity ≠ null then
            RESUME(activity)
```

Listing 4.14: Task switch finalisers of the monitor wait operation.

6. If there is no other activity ready to run, the ownership is released and an ordinary cooperative task switch is performed. This can only happen when all other processors perform a task switch at the same time in which case the ready queue is about to get filled immediately.

7. As soon as a waiting activity is resumed, it tries to acquire the monitor using an ordinary call to the ENTER procedure providing the nesting level stored beforehand. If the ownership was transferred to this activity by the TRANSFERAWAIT task switch finaliser, the corresponding acquisition will always succeed. This ensures that the current activity has always the ownership when returning to the compiler generated code that rechecks the previously unsatisfied await condition.

Our monitor implementation improves upon the existing implementation in A2 in several ways. In A2, await conditions are always checked by the current owner of the monitor instead of the waiting activity [Mul02]. Our implementation in contrast performs task switches to waiting activities and therefore always evaluates await conditions in the correct context.

Furthermore, the original Active Oberon report does not allow activities to enter an exclusive region more than once [Rea04]. This restriction always seemed somehow artificial and misled many developers into duplicating procedures once using exclusive regions and once without. Our implementation derestricts reentrant exclusive blocks which simplifies the corresponding implementation when blocked and waiting activities transfer the ownership to each other. A proposal for the relaxation of the corresponding language rules concerning reentrancy in Active Oberon is described in Section C.14 on page 216.

4.4 Multiprocessor Support

The scheduler and synchronisation primitives described so far are completely portable and do not require any special architectural support. Regarding the implementation of multiprocessors however, different shared-memory architectures may impose distinct requirements on the runtime system. This concerns issues like properly setting up cache coherency protocols in order to comply with our memory model, or enabling multiprocessing using physical processors, multiple cores, or hardware multithreading.

In order to assemble all inherently non-portable hardware dependencies regarding multiprocessing in a single place, we tried to minimise the number of functions that need to be implemented anew for each targeted architecture. This set of functions acts therefore as hardware abstraction layer for the multiprocessor support of the runtime system. Its main purpose is basically to start and stop the scheduler on each of the logical processors of the machine and consists of the following three parameterless procedures:

- StartAllProcessors

  This procedure is called once when the runtime system is initialised and starts the scheduler on each logical processor of the system. Its main purpose is to initialise processors uniformly and to count the total number of processors which is required for all lock-free algorithms that make use of processor-local storage.

Within the kernel for the AMD64 architecture for example, this procedure uses the advanced programmable interrupt controller (APIC) in order to send an interprocessor interrupt to all other logical processors in the system. This includes physical processors as well as all of their cores and hardware threads.

Beforehand, it sets up simplified bootloaders which are executed by each logical processor when they receive the initial interprocessor interrupt. The task of the bootloader is to jump from real mode (16-bit) over protected mode (32-bit) to long mode (64-bit) and initialise the processor accordingly before starting the scheduler in the end.

In the case where the runtime system is running as an application library on top of Windows or Linux based operating systems, this procedure starts ordinary threads instead. The basic idea is to emulate all logical processors of the underlying machine by starting one thread per processing unit. By assigning distinct thread affinities, we can actually ensure that each thread is executed only on its corresponding processor.

- SUSPENDCURRENTPROCESSOR

  This procedure is used to temporarily disable logical processors that have no work to do. The runtime system maintains a set of idle activities, one per logical processor, which all call this procedure when there is no other activity ready to run.

  In the native case, this procedure basically executes a hardware-dependent sequence of instructions which puts the processor in a low power state. In this state, the processor halts its execution and waits for the next interrupt to occur.

  Regarding the case where processors are represented by ordinary threads of the underlying operating system, this procedure makes use of the synchronisation primitives provided by the system. In Windows for example, a global semaphore counting the number of non-idle activities is used for this purpose.

- RESUMEANYPROCESSOR

  The task of this procedure is to wake up any logical processor that is currently suspended because of a call to the SUSPENDCURRENTPROCESSOR procedure. The scheduler calls this procedure whenever a new activity enters the system or a currently suspended one gets resumed, see for example Listing 4.5 on page 75.

  In the native kernel, this procedure sends an interprocessor interrupt to a suspended processor which subsequently resumes its execution. Where idling processors are implemented using a global semaphore provided by the underlying operating system, this procedure signals the semaphore instead.

4.5 Interrupt Handling

The goal of the runtime system was to be independent of the hardware architecture, and is therefore designed not to require any support for hardware interrupts for its own implementation. Where the runtime system is used as a kernel for an operating system however, it still provides the necessary means to write drivers for interrupt-driven devices.

In this section we describe a lock-free and portable interrupt handling model which allows to view device drivers as cyclic processes which can just wait for the next interrupt to occur. This model is similar to the alternative one proposed by Muller for the Active Object System [Mul02], but uses a library approach instead of special statements built into the programming language. The reason for this design is the observation that the actual number of an interrupt request is highly platform specific and should not be bundled with a high-level language specification.

The interrupt handling mechanism is based on the idea that there are no designated processes responsible for handling interrupts. Any activity in the system should ideally be able to await the occurrence of an interrupt. Interrupts are therefore modelled like any other synchronisation primitive and can thus use the same framework provided by the scheduler.

Listing 4.15 on the facing page shows the definition and initialising operations of the data structure called INTERRUPT that can be used by implementers of device drivers to wait for an interrupt. It consists of the unique number of the actual interrupt request and a local time stamp value which indicates when the corresponding interrupt was handled for the last time. The number of the interrupt request can be used as index into the global arrays shown after the data structure definition. These arrays store a queue of activities waiting for the corresponding interrupt as well as the global time stamp value which is incremented whenever an interrupt occurs.

Before users can wait for an interrupt, the corresponding data structure must be initialised and the actual interrupt request has to be enabled. The corresponding operation to install an interrupt is called INSTALL. This procedure copies the interrupt index and the current value of the corresponding time stamp into the user-provided data structure. The time stamp must be queried using an atomic compare-and-swap operation in order to read consistent data in case the time stamp is modified concurrently.

Afterwards, a low-level interrupt handler routine is installed using a machine-dependent procedure called INSTALLINTERRUPT. This procedure is the only operation of the interrupt handling model that is not portable as it is highly hardware specific. Its main purpose is to drive the interrupt controller or equivalent devices in order to unmask the corresponding interrupt. In addition, it modifies the interrupt vector table such that the provided interrupt handler gets called whenever the interrupt occurs. It also ensures that the context of the interrupted activity is restored properly after the interrupt handler returns.

The actual interrupt handler is called HANDLEINTERRUPT shown at the bottom of Listing 4.15. It increments the global time stamp associated with the interrupt and resumes all waiting activities. Any overflow while incrementing the time stamp can be ignored as the time stamps are only compared for equality. Before calling the uncooperative operations of the queue and the scheduler however, the interrupt handler temporarily swaps the currently executing activity. This is necessary in order to comply with the semantics of uncooperative blocks which require that only one activity per processor is executing code inside an uncooperative block.

```
structure INTERRUPT
    index : integer
    timestamp : integer

variable
    timestamps : array Interrupts of integer
    awaitingQueue : array Interrupts of QUEUE
    virtualProcessor : array Interrupts of ACTIVITY

procedure INSTALL(interrupt, index)
    interrupt.index ← index
    interrupt.timestamp ← CAS(timestamps[index], 0, 0)
    INSTALLINTERRUPT(HANDLEINTERRUPT, index)

procedure HANDLEINTERRUPT(index)
    uncooperative
        current ← GETACTIVITY()
        SETACTIVITY(virtualProcessor[index])
        timestamp[index] ← timestamp[index] + 1
        activity ← DEQUEUE(awaitingQueue[index])
        while activity ≠ null do
            RESUME(activity)
            activity ← DEQUEUE(awaitingQueue[index])
        SETACTIVITY(current)
```

Listing 4.15: Data structures for interrupt handling.

Since interrupts are never explicitly disabled, they can basically occur at any time and could therefore also happen during the execution of an uncooperative block. If the corresponding interrupt handler in turn executes the very same uncooperative block, there are theoretically speaking two activities executing the same block. This has disastrous consequences for lock-free operations using processor-local storage which highly depend on uncooperative blocks to be executed entirely without being interrupted by another activity.

Our solution is to change the currently executing activity temporarily to a special-purpose activity associated with the interrupt. We call the corresponding activity a virtual processor because it stores a unique processor index which differs from all physical processors in the system. Using this trick, lock-free operations cannot distinguish a processor executing an interrupt handler from any other processor as explained in Section 2.3.1.

Although this design allows nested interrupts in general, it implies that a handler for one particular interrupt may not interrupt itself. However, this behaviour is hardly required in practice and can be prevented completely when hardware interrupts have to be acknowledged first to the interrupt controller. Thanks to this convention, comparing time stamps is sufficient in order to ensure that interrupts are never missed.

Once an interrupt data structure is initialised and the corresponding hardware interrupt is enabled, device drivers can wait for an interrupt using the AWAITINTERRUPT procedure showed in Listing 4.16 on the next page. It first compares the local time stamp of the interrupt data structure against the value of the global time stamp. If the two values do not match, there was an interrupt and the procedure returns immediately by updating the local time stamp value.

Otherwise it selects any activity ready to run and performs a task switch to that activity. The corresponding task switch finaliser called ENQUEUE-INTERRUPT places the suspended activity into the waiting queue associated with the corresponding interrupt. As always, the value of the time stamp must be rechecked in order to prevent activities from remaining suspended forever because the interrupt could have already occurred during the task switch or the enqueue operation.

```
procedure AWAITINTERRUPT(interrupt)
    uncooperative
        index ← interrupt.index
        timestamp ← CAS(timestamps[index], 0, 0)
        while interrupt.timestamp = timestamp do
            repeat
                activity ← SELECT(Idle)
            until activity ≠ null
            SWITCHTO(activity, ENQUEUEINTERRUPT, interrupt)
            FINALIZESWITCH()
            timestamp ← CAS(timestamps[index], 0, 0)
        interrupt.timestamp ← timestamp

procedure ENQUEUEINTERRUPT(previous, interrupt)
    uncooperative
        index ← interrupt.index
        timestamp ← interrupt.timestamp
        ENQUEUE(previous, awaitingQueue[index])
        if CAS(timestamps[index], 0, 0) ≠ timestamp then
            activity ← DEQUEUE(awaitingQueue[index])
            while activity ≠ null do
                RESUME(activity)
                activity ← DEQUEUE(awaitingQueue[index])
```

Listing 4.16: Procedure for awaiting the occurrence of an interrupt and the corresponding task switch finaliser.

Our interrupt handling mechanism is lock-free and uses only lock-free data structures. The main advantage of this approach is that we can make use of a very important property of non-blocking algorithms. Neither the lock-free operation executed by the interrupt handler nor the interrupted activity which is potentially operating on the same data structure wait for each other. If a lock-free operation gets interrupted by an invocation of the same operation, it just looks like another processor was faster executing the corresponding atomic operations. Whether the interrupted code was overtaken by another physical or the same processor does not matter at all. This is why lock-free algorithms are in general considered reentrant by design. Lock-free data structures are therefore very well suited for being used in interrupt handlers. In particular, non-blocking synchronisation in interrupt handlers is what makes lock-freedom even beneficial for machines with only one processor.

5 Lock-Free Memory Management

Our lock-free queue used by the scheduler potentially allocates memory during enqueue operations. In order to call the scheduler lock-free, the corresponding memory allocation must therefore be lock-free as well. This chapter describes our approach towards a simple and portable lock-free memory management including a concurrent garbage collector that completes the required runtime support for Active Oberon.

5.1 Heap Management

Lock-free memory managers have not been investigated as thoroughly in the standard literature as other lock-free data structures such as queues. Two of the few known implementations have been presented concurrently and independently by Michael [Mic04c] as well as Gidenstam and Papatriantafilou [GPT05]. Both approaches are based on a well-known and practical concurrent memory allocator called Hoard [BMBW00, Ber02] and use atomic compare-and-swap operations in order to create lock-free versions of it. Dice and Garthwaite on the other hand provided a memory allocator which is lock-free for the most part [DG02].

As all of these memory allocators are derived from Hoard, they classify memory allocations according to the requested size and dedicate large chunks of memory called superblocks for this purpose. Interestingly, Hoard only manages memory blocks that fit into superblocks and passes the allocation of larger blocks and superblocks themselves on to the virtual memory management of the underlying operating system. However, as long as the virtual memory management system is not implemented in a lock-free manner as well, any memory management built on top of it does strictly speaking not qualify as a lock-free algorithm.

Since our runtime system does not rely on virtual memory management or separate address spaces, we have the advantage of representing the complete physical memory as one contiguous superblock referred to as a heap. In order to allow heaps to encompass an arbitrary memory area, we decided to represent allocated and free memory blocks using a lock-free binary buddy system. Buddy systems are well-known memory allocation schemes that are fast and expose little external fragmentation [Kno65, Knu68]. In the context of lock-free programming, they offer an additional distinct advantage over other dynamic allocation schemes. While other techniques have to look up and modify several addresses simultaneously while coalescing adjacent free memory blocks, buddy systems can compute the addresses instead and have to modify only single memory locations. This is important for portably merging free memory blocks into bigger ones on machines that only provide single compare-and-swap operations.

Our design allows to manage several heaps which are represented as contiguous memory ranges of arbitrary size. Each heap is partitioned into blocks where each block size is a power of two. For this purpose, the heap maintains for each block size 2^i a singly linked list of free blocks called $free_i$. These lists are accessed as stacks whereby an allocation of a memory block basically pops a block from the free stack, and a deallocation pushes it again onto the stack.

The data structures representing heaps and memory blocks are shown in Listing 5.1 on the facing page. A heap consists of a contiguous sequence of bytes addressed by $[begin \ldots end)$ within which all managed memory blocks reside. This assumes that heaps are non-overlapping and manage distinct memory ranges. As stated above, each heap also maintains an array of stacks which point to the first element of a linked list of free blocks with the same size. In addition, there is an array of stacks containing blocks which cannot be deallocated safely and are therefore pooled separately.

The number of different block sizes depends on the size of the actual memory range but cannot exceed $\lfloor \log_2(end - begin) \rfloor$. Regarding a heap encompassing the complete available address space, the maximal number of required block sizes therefore equals the actual bit width of the address size of the underlying machine.

```
structure HEAP
    begin, end : address
    free : array address bit width of pointer to BLOCK
    pooled : array address bit width of pointer to BLOCK

structure BLOCK
    index : integer
    next : pointer to BLOCK
```

Listing 5.1: Data structures of lock-free heaps and their memory blocks.

Each allocated memory block stores its size as index of the corresponding stack. If the memory block is free and therefore part of a linked list of free blocks, it stores a reference to its successor in that list. This datum is completely ignored and reused for the actual payload in case the memory block becomes allocated. The minimal block size of allocated and free blocks is therefore two words. In practice, we use a multiple of this block size in order to compensate for the overhead of the meta data.

Before a heap is available for allocation and deallocation of memory blocks, its data structure has to be initialised properly. At the beginning, an empty heap consists of the minimal amount of free blocks that is required to cover the complete memory range. The corresponding initial partitioning of the heap is performed by the INITIALIZE procedure shown in Listing 5.2 on the next page. Since we assume that this operation is performed once at the beginning by a dedicated process, there is no need to protect the code from concurrent access.

The procedure consecutively appends the biggest block not exceeding the remaining free space to the corresponding stack. The biggest block is found by halving a block whenever its buddy block adjacent to it would completely lie outside the specified memory range. If the last free block is found, all stacks with a smaller block size are initialised to be empty. This algorithm assumes that the specified address range is aligned to the minimal block size and has space for at least one block.

```
procedure INITIALIZE(heap, begin, end)
    heap.begin ← begin
    heap.end ← end
    index ← ⌊log2(end − begin)⌋
    repeat
        next ← begin + 2^index
        while next > end do
            index ← index − 1
            heap.free[index] ← null
            heap.pooled[index] ← null
            next ← begin + 2^index
        block ← begin
        block→next ← heap.free[index]
        heap.free[index] ← block
        heap.pooled[index] ← null
        begin ← next
    until begin = end
    while index ≠ 0 do
        index ← index − 1
        heap.free[index] ← null
        heap.pooled[index] ← null
```

Listing 5.2: Initialisation of a memory heap partitioning it into the smallest possible amount of free blocks.

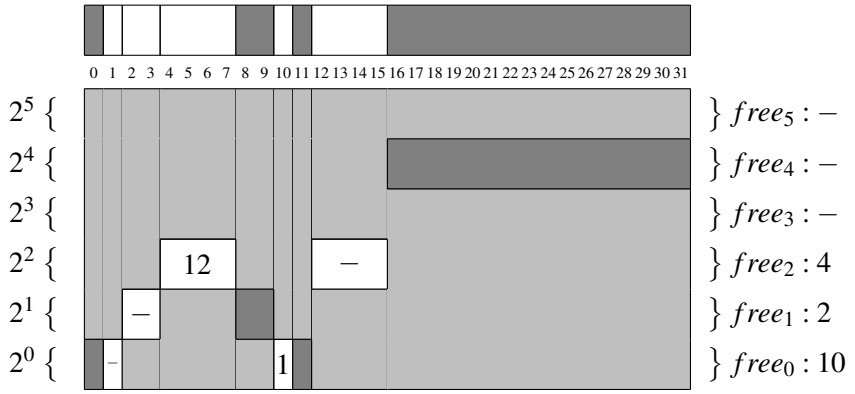

Figure 5.1: Sample memory allocation in a heap of size 32.

An example of a heap managing up to 32 blocks is depicted in Figure 5.1. For illustrative purposes, the unit of an address in this example is equal to the minimal block size of the system. The sample heap therefore manages blocks in the address range $[0 \ldots 32)$. It consists of four allocated blocks which are shaded grey in the heap layout depicted at the top of the figure. The addresses of the allocated blocks are 0, 8, 11, and 16. The remaining white blocks are free for allocation and are chained in a linked list that begins with the top of the stack of the corresponding size as shown on the right hand side. Regarding block size 4 for example, there are two free blocks at addresses 4 and 12 as indicated by the linked list referenced by the stack $free_2$.

An allocation in our buddy system first tries to find one of the smallest free blocks providing enough space for the requested size. A free block is found and allocated quickly by inspecting and popping the top of the corresponding free stack. In a second step, the resulting block is halved into two adjacent blocks as long as the requested memory still fits into one half. During this operation, the other half is deallocated and pushed back to the corresponding free stack.

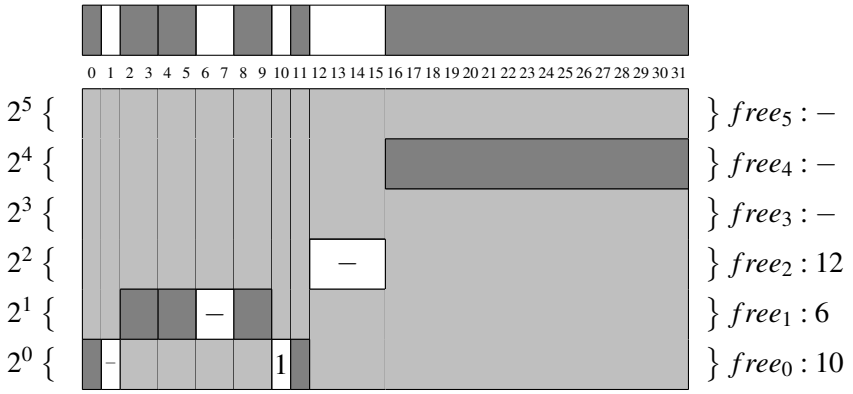

Figure 5.2: Same heap after two consecutive block allocations of size 2.

If for example a block of size 2 is requested in the example depicted in Figure 5.1, the block at address 2 would be popped from stack $free_1$. A subsequent allocation of the same size would try to pop a block from stack $free_2$ instead since $free_1$ is now empty. The right half of the resulting block with address 4 is deallocated again and pushed onto stack $free_1$. The resulting layout of the heap and its data structures is depicted in Figure 5.2.

Memory deallocations in the buddy system just reverse the operations of a memory allocation. Each deallocation of a block checks whether the corresponding buddy of the block is free or currently allocated. Two adjacent free buddy blocks are coalesced into one bigger block which can potentially be merged again with its own buddy recursively. Ideally, the deallocation completely reverts the partitioning from beforehand yielding exactly the same heap layout as before.

One issue of representing free blocks using lock-free stacks however, is the fact that naive implementations of the pop operation are prone to the ABA problem: If a stack ends up having the same element at the top after several push and pop operations, a delayed pop operation does not detect any change and may erroneously replace the top with an outdated successor.

```
variable
    hazard : array N of pointer to BLOCK

procedure ACCESS(reference to shared)
    uncooperative
        block ← CAS(shared, null, null)
        repeat
            hazard[processor] ← block
            block ← CAS(shared, null, null)
        until block = hazard[processor]
        return block

procedure ISHAZARDOUS(block)
    uncooperative
        index ← 0
        while index ≠ N ∧ block ≠ hazard[index] do
            index ← index + 1
        return index ≠ N
```

Listing 5.3: Data structure and operations for managing hazard pointers.

The ABA problem is also an issue in related work where it is generally solved using pointer tagging [Mic04c, GPT05]. As explained in Section 3.4 however, this technique may be sufficient to solve the problem in practice but is conceptually unsound because pointer tags could overflow and wrap around. Based on the sound solution to the ABA problem of our lock-free queue, our heap manager makes use of hazard pointers instead [Mic04b].

Similar to the lock-free queue implementation, the heap manager uses processor-local storage in order to represent hazard pointers. Listing 5.3 shows the corresponding global array for N processors. The subsequent ACCESS procedure returns the current value of a shared memory block reference which has been marked as hazardous. The ISHAZARDOUS procedure returns whether a memory block reference matches any hazard pointer.

```
procedure POOL(block, index, heap)
    uncooperative
        repeat
            pooled ← CAS(heap.pooled[index], null, null)
            block→next ← pooled
            value ← CAS(heap.pooled[index], pooled, block)
        until value = pooled

procedure FREE(index, heap)
    uncooperative
        pooled ← CAS(heap.pooled[index], null, null)
        if CAS(heap.pooled[index], pooled, null) = pooled then
            while pooled ≠ null do
                next ← pooled→next
                RELEASE(pooled, index, heap)
                pooled ← next
```

Listing 5.4: Operations for managing pooled memory blocks.

A memory block is marked as hazardous whenever it is dereferenced in order to read its successor which is used to pop the block from the corresponding free stack. A hazardous memory block implies that it is still in use and may therefore not be pushed back onto the free stack. This prevents concurrent pop operations from seeing the same block twice at the top of the stack after it has been already allocated temporarily which effectively solves the ABA problem. The deallocation of hazardous memory blocks may therefore not take place immediately and has to be deferred.

Hazardous memory blocks can be pooled for deallocation by pushing them to a completely different stack called *pooled*. The corresponding lock-free operation to pool a single hazardous memory block is called POOL and is shown in Listing 5.4. It just pushes the memory block to the corresponding stack by repeatedly trying to atomically replace the top element with the pooled block.

```
procedure ACQUIRE(index, heap)
    uncooperative
        FREE(index, heap)
        repeat
            free ← ACCESS(heap.free[index])
            if free = null then
                hazard[processor] ← null
                return null
            next ← free→next
            value ← CAS(heap.free[index], free, next)
            hazard[processor] ← null
        until value = free
        return free
```

Listing 5.5: Lock-free acquisition of blocks from the specified free stack.

Pooled memory blocks can be deallocated using the FREE procedure. Rather than popping a single element from the stack which is again not safe from the ABA problem, this procedure atomically exchanges the reference to the block at the top with an invalid reference. If this operation succeeds, the procedure gets exclusive access to the linked list of pooled memory blocks and all subsequent pool operations operate on an empty stack. The linked list is then traversed and each block gets deallocated in turn.

The actual allocation and deallocation of memory blocks is performed using two helper functions called ACQUIRE and RELEASE. They are named like this because an activity allocating a memory block conceptually gets exclusive ownership over it.

The algorithm for acquiring a heap block from a specified free stack is shown in Listing 5.5. It basically resembles a lock-free pop operation because it tries to remove the block at the top from the specified free stack. It additionally marks this block as hazardous using a call to the ACCESS procedure in order to read its successor safely. At the very beginning, it first frees all memory blocks that have been pooled to defer their deallocation.

```
procedure RELEASE(block, index, heap)
    uncooperative
        repeat
            loop
                if ISHAZARDOUS(block) then
                    POOL(block, index, heap)
                    return
                buddy ← heap.begin + (block − heap.begin) ⊕ 2^index
                free ← ACCESS(heap.free[index])
                if free ≠ buddy then
                    hazard[processor] ← null
                    break
                next ← free→next
                value ← CAS(heap.free[index], free, next)
                hazard[processor] ← null
                if value ≠ free then
                    free ← value
                    break
                index ← index + 1
                if buddy < block then
                    block ← buddy
            block→next ← free
            value ← CAS(heap.free[index], free, block)
        until value = free
```

Listing 5.6: Lock-free release of blocks to the specified free stack.

The corresponding lock-free release operation is shown in Listing 5.6. It first checks whether the heap block given as argument is hazardous in which case it gets pooled and the procedure returns immediately. Otherwise, it computes the address of the buddy of the block by swapping the address bit that corresponds to its size. The procedure then tries to allocate the buddy by performing a lock-free pop operation on the corresponding free stack.

```
procedure ALLOCATE(size, heap)
    uncooperative
        index ← ⌈log2(size + displacement)⌉
        current ← index
        block ← ACQUIRE(current, heap)
        while block = null do
            current ← current + 1
            block ← ACQUIRE(current, heap)
        while current ≠ index do
            current ← current − 1
            RELEASE(block + 2^current, current, heap)
        block→index ← index
        return block + displacement

procedure DEALLOCATE(address, heap)
    uncooperative
        block ← address − displacement
        RELEASE(block, block→index, heap)
```

Listing 5.7: Lock-free heap allocation and deallocation.

As before, the top of the stack is marked as hazardous in order to protect this operation from the ABA problem. If the buddy could be allocated, both memory blocks are merged and the complete operation is repeated with the coalesced block. If there are no more merges possible, the resulting memory block is released by pushing it onto the corresponding free stack. The push operation fails if another activity concurrently changed the stack in which case the whole procedure starts anew.

Finally, the actual heap allocation and deallocation operations provided to users of the heap are called ALLOCATE and DEALLOCATE and are shown in Listing 5.7. A memory allocation takes an arbitrary size and returns the first address of a memory area which can be deallocated once afterwards.

The allocation consists of two consecutive loops. The first loop tries to acquire a block with the required size by incrementing the index until a free block is found. The second loop repeatedly releases the right half of the acquired block as long as the required memory fits in the other half. The returned value is the address of the block displaced by the size of the meta data used for storing the index of the block. The deallocation on the other hand subtracts this displacement from the address given as argument in order to retrieve a pointer to the corresponding memory block which then can just be released.

Our approach is lock-free as we only access lock-free stacks while maintaining the set of free blocks. Aside from starvation issues, faulty processes that allocate or deallocate memory do therefore not compromise the progress of other processes performing the same operations. However, a faulty process may cause acquired memory to get lost forever if that process has the ownership over an acquired block. One can ignore this issue, since the same applies also if the process fails to call the DEALLOCATE procedure in the first place.

One drawback of our approach is that only the topmost block of the free stack is considered while coalescing adjacent blocks. Therefore, the probability that a block can be merged with its buddy correlates negatively with the amount of free blocks in the stack for the corresponding size. A solution to this issue is to temporarily get ownership over the complete stack in order to search for the buddy in the linked list of free blocks. The ownership can be granted by atomically exchanging the reference to the first block in the linked list with an invalid reference. This allows to traverse the list and remove the corresponding block without requiring any protection from concurrent access. During this period, allocations with the same block size will allocate bigger blocks and halve them as usual. In order to compensate this overhead per deallocation, one can also create a concurrent activity which periodically gets the ownership of one stack at a time and coalesces all adjacent blocks in that list.

The heap management shown in this section is used by the runtime system when it is deployed as a kernel of an operating system. In this case, the managed heap consists of the complete main memory excluding the code

and data of the statically linked kernel itself. However, the physical memory layout of the underlying machine often consists of so-called memory holes which describe areas in the physical address space that are not mapped to main memory. In order to accommodate these memory holes, the runtime system creates several individual heaps to cover all of the accessible main memory. Another solution could use a single heap instead but extend its implementation with a special-purpose operation that allows to exclude certain memory areas from being allocated.

5.2 Stack Management

In a system capable of multithreading, each thread of execution typically has its own contiguous stack memory it can operate on. In order to allow a large number of concurrent threads, operating systems tend to minimise the amount of memory set aside for each stack. A popular approach implemented by A2 and other systems based on virtual memory management is to reserve several consecutive pages of virtual memory for a stack but only associate them with a few pages of physical memory [Mul02, RSI12]. This technique enables relative small initial stacks but allows to grow them dynamically if threads require more stack memory later on. A request to grow the stack dynamically is indicated when a memory access inside the reserved virtual memory leads to a page fault. A further advantage of this approach is the automatic check for stack overflows which is indicated by a memory access outside the reserved virtual memory region.

Another solution consists of statically examining the call graph of a thread and noting the maximal amount of required stack space [Die92]. This approach allows to allocate the exact amount of stack memory for each individual thread which can be substantially smaller than the page size of the underlying machine. This technique has the advantage that its implementation is portable since it does not have to rely on virtual memory because stack overflows cannot occur. In general however, it is impossible to determine the actual call graph if the code makes use of recursion or indirect function calls and corresponding runtime checks are still required.

Alternatively, dynamic stack management can also be achieved using compiler-instrumented procedures that allocate an additional block of stack in their prologue whenever required [Blä07]. All heap-allocated stack blocks form a linked list which is cleared when procedures return and no longer use the surplus stack memory. This allows processes to begin with a very small preallocated stack which grows as needed but still allows a large amount of concurrent processes in the system.

Our approach aims at portability and is a mixture of all three ideas discussed above. Instead of examining the complete call graph, it only relies on the amount of required stack memory per procedure call. This number is statically known by the compiler and already required when it generates the code to create a new stack activation frame in a function prologue. Our compiler adds a stack check at the beginning of each procedure similar to checks generated for implicit cooperative multitasking as explained in Section 4.2.4. We chose the stack check to be performed by the callee because functions are usually called more than once.

Thanks to our approach to cooperative multitasking, the descriptor to the currently running activity is handily available in a dedicated general-purpose register. The descriptor contains a pointer to the beginning of the stack as shown in Listing 4.1 on page 71. In addition, it also contains a pointer to the stack limit which specifies the memory range within which stack accesses are valid. A stack overflow is indicated when the creation of a stack activation frame exceeds this limit. In this case, the generated check calls a procedure provided by the runtime system to handle the overflow.

Listing 5.8 on the next page shows the corresponding instruction sequence of two sample stack frame creations as generated by the compiler targeting the AMD64 hardware architecture [Adv13]. As shown on the left hand side, the compiler normally activates a stack frame by storing the former frame pointer on the stack and subtracting the size of the new stack frame from the stack pointer. The stack check on the right hand side uses a temporary register called `rax` instead and compares its value against the limit stored in the descriptor referenced by `rcx`. If a stack overflow is detected, the EXPANDSTACK procedure gets called which grows the stack dynamically and returns the new stack pointer in `rax`.

```
; default prologue
  push rbp
  mov  rbp, rsp
  sub  rsp, 200
```

```
; checked prologue
  push rbp
  mov  rbp, rsp
  lea  rax, [rsp - 200]
  cmp  rax, [rcx + 80]
  jae  skip
  push rax
  call ExpandStack
skip:
  mov  rsp, rax
```

Listing 5.8: Generated instruction sequence for default and checked function prologues requesting 200 bytes as stack activation frame.

The stack is expanded by allocating new stack memory from the heap and copying the contents from the old stack to the new one. The size of the new stack is determined by doubling the size of the old stack until the requested stack pointer would be covered. This exponential growth allows to compensate the overhead of allocating and copying the stack. If the new size exceeds a predefined upper limit, the current activity aborts with a stack overflow error. Otherwise, a new memory block of this size is allocated and the complete contents of the old stack are copied over. This copy process requires to adapt all references to local variables. Regarding Active Oberon, this only affects variable parameters for which the compiler provides the necessary meta data. The old stack is discarded after its complete contents are copied into the new increased stack. Finally, the stack pointers of the current activity are adapted to point into the new stack.

The actual stack limit is slightly less than the allocated stack in order to reserve some guaranteed stack space that is always available. This allows a procedure to elide the stack check completely if the compiler can statically determine that a call of this procedure does not require more stack memory than reserved. This is an optimisation in general, but is actually crucial for all procedures involved in the actual expansion of the stack in order to prevent an infinite recursion.

The whole process of checking and expanding is lock-free because each activity operates on its own local stack which does not affect the progress of any other activity in the system. New stacks are created using the standard lock-free memory allocation presented in the previous section.

5.3 Garbage Collection

Active Oberon and its ancestor Oberon both require automatic memory management in the form of a garbage collector [WG92, Rea04]. In A2 as well as in the Oberon System, the garbage collector is a dedicated process with first class citizen status [Mul02]. In both systems, processes that change the object graph, so-called mutators, are not allowed to run while the dedicated process, the collector, is retrieving unused memory. This behaviour is generally known as stop-the-world garbage collection and notorious for causing arbitrarily long and non-deterministic delays. This affects all processes running on the system regardless of whether they are currently allocating memory or not. In particular, the unpredictable delays caused by the runtime system prevents any program running on top of it from satisfying real-time constraints.

However, the most important issue of stop-the-world garbage collection with respect to the topic of this thesis is its blocking behaviour. Since mutators have to wait for collectors to complete their task, the exclusive access to the object graph by the garbage collector is tantamount to a global lock. As a result, this design inherits all the issues of blocking synchronisation as described in Section 2.1. Most importantly, there is no guaranteed progress of the system because a faulty collector could cause mutators to wait forever.

An obvious improvement over this kind of blocking is to design the garbage collector to perform its task incrementally. If the collector marks and sweeps objects step by step rather than in one go, it basically cannot cause arbitrary long delays [BA84]. Regarding the Oberon system, Bianchi has created an incremental garbage collector running on uniprocessor systems [Bia98].

A more robust solution however, is to design the automatic memory management to be lock-free which actually implies incremental operation and provides even stronger guarantees. Herlihy and Moss were amongst the earliest researchers to pursue this approach [HM91]. However, their collector relies heavily on copying for moving and updating objects which renders their approach inappropriate for practical purposes. Gao et al. presented one of the first lock-free garbage collectors based on mark and sweep [GGH07]. Other authors have focused their research on lock-free reference counting instead [Val94, HG01, HLMM02, DMSJ02]. Gidenstam gives an excellent overview over this topic [GPST09]. Reference counting seems to be appropriate and useful for managing the objects participating in lock-free data structures. On the other hand, this mechanism is known to be vulnerable to cyclic references which hinder objects from being reclaimed [MS95] and therefore is not suited for a general-purpose memory management. Unfortunately however, many authors based their work on atomic double compare-and-swap operations which are not universally available on contemporary hardware [DMSJ02, HLMM02].

The main goal of our implementation of the garbage collection was to guarantee the progress of mutators in a lock-free manner. Abolishing the global lock implies that collectors lose their first class citizen status and must be modelled like any other ordinary activity running concurrently in the system. In fact, our design goes one step further by allowing the runtime system to refrain from garbage collection altogether if desired, rendering automatic memory management completely optional and independent. For this reason, we introduced a new language feature for manual memory deallocation in case a program does not want to rely on garbage collection, see Section C.8 on page 210. The runtime system itself manages all of its resources on its own by making use of this feature and does therefore not require any automatic memory management at all. It is therefore very well suited for execution environments where garbage collection might impose an inappropriate overhead.

Enforcing that mutators never wait for collectors basically implies that they have to be able to execute concurrently. Concerning our runtime system, this requirement has two additional important implications:

- Regarding a single-processor system, a collector must yield the control of execution to a mutator in a regular fashion in order to maintain the illusion of parallelism. A garbage collection must therefore be able to be paused and execute incrementally. In particular, a collector has to be prepared for mutators to modify the object graph concurrently.

- The set of objects that can be garbage collected consists of all objects that are not reachable directly or indirectly from the root set. In the case of Active Oberon, the root set consists of all objects referenced directly by global and local variables. The set of global variables is static and known in advance, whereas the set of local variables depends on the number of running and suspended activities in the system and their respective call stacks.

  Disallowing a collector to temporarily suspend an activity while traversing its call stack requires special attention when the activity runs concurrently and constantly modifies the stack. The same applies to suspended activities that resume their execution while the collector is examining their stack in parallel.

Both issues are related and basically require mutators to cooperate with the garbage collector. Our approach introduces so-called write barriers which cause the compiler to instrument assignments to reference variables with special code. In the case of our runtime support for Active Oberon, the actual task of a write barrier is threefold:

1. The referenced object that is assigned to the variable gets marked and is therefore not available for reclamation. A marked object is added to a linked list of objects that need to be traced for outgoing references. This is necessary if the garbage collector has already visited the variable while marking the object graph but the object becomes only reachable through this variable in the meantime. By traversing the linked list of marked objects in the end, the garbage collector still reaches objects which otherwise would have been be collected.

2. A processor-local flag is set temporarily in order to indicate that the current processor is modifying a variable. The garbage collector has to traverse the linked list of marked objects as long as concurrent assignments are ongoing on other processors in order to ensure that all marked objects are eventually traced for outgoing references. Thanks to cooperative scheduling which enables processor-local storage, this effectively synchronises collectors with mutators without requiring the latter to wait for the former.

3. If the modified variable is on the stack of an activity, the write barrier also updates a reference counter stored in the referenced object. This counter keeps track of the number of local variables referring to the object. A non-zero value of this counter indicates that an object is still referenced by a running or suspended activity and therefore belongs to the root set. Because only local variables are reference-counted, there cannot be any reference cycles and what is more, collectors do never have to inspect call stacks explicitly.

Write barriers naturally incur some overhead in execution time. However, the implicit generation of write barriers can be disabled completely by using a corresponding compiler flag. This is useful when targeting execution environments that do not provide a garbage collector and require applications to manage memory on their own.

In cases where applications depend on garbage collection on the other hand, the write barriers cannot be elided because they are imperative to detect the set of objects that are still reachable. Since assignments imply the marking of an object, write barriers require the garbage collector to be present and to provide the corresponding functionality. The garbage collector uses the following three lock-free data structures for this purpose:

- An atomic counter called cycle count is incremented at the beginning of each garbage collection. In addition to this global counter, each object stores its own copy of the cycle count which acts as the mark bit. It identifies the garbage collection cycle which was the last one that could reach the object while marking.

A collector marks objects by updating their cycle count if it is less than the current value of the global counter. In the end, it can safely collect objects that have a cycle count that is not greater than the value which the global counter had before is was incremented. By using signed arithmetic, the garbage collector subtracts cycle counts from each other while comparing them in order to deal with potential increment overflows causing cycle counts to wrap around.

- A linked list called watched list contains all objects that are managed by the garbage collector. Each object has a reference to its successor in this list and gets automatically inserted by the runtime system during its creation. The managed unit of the garbage collector are therefore arbitrary objects in contrast to allocated memory blocks. In principle, the garbage collector could thus not only manage memory but also any other resource.

  The main purpose of this data structure however, is the complete detachment of the garbage collector from the underlying heap manager. The garbage collector is therefore also able to handle objects that are allocated by the heap manager of the underlying operating system if the runtime system is deployed as an application library.

- A linked list called marked list contains all objects that have been marked but not yet scanned for outgoing references. Each object has an additional reference to its successor in this particular list. A mark operation updates the cycle count of the object if necessary and puts it into the list of marked objects. A subsequent trace operation traverses the marked list and marks all outgoing references.

  This data structure allows to break the implicit recursion while marking and tracing the object graph by splitting it into two separate operations. The main advantage is that mutators can cooperate with the garbage collector by marking objects during assignments without having to trace them afterwards. In addition, several collectors can help each other by concurrently tracing and marking the shared list of marked objects.

```
procedure MARK(object)
    uncooperative
        repeat
            cycle ← CAS(object→cycle, 0, 0)
            current ← CAS(cycleCount, 0, 0)
            if cycle − current ≥ 0 then
                return
            value ← CAS(object→cycle, cycle, current)
        until value = cycle
        loop
            first ← CAS(firstMarked, null, null)
            if CAS(object→nextMarked, null, first) ≠ null then
                return
            if CAS(firstMarked, first, object) = first then
                return
            CAS(object→nextMarked, first, null)
```

Listing 5.9: Lock-free mark operation.

Furthermore, the marked list basically resembles a data structure called mark stack known from traditional garbage collectors. This representation of the mark stack however can grow arbitrarily without ever requiring any additional memory and is even able to contain all objects of the object graph in the worst case.

The actual lock-free MARK procedure is shown in Listing 5.9. It is called concurrently by collectors marking the root set as well as by write barriers of mutators modifying reference variables. The procedure first tries to atomically update the cycle count of the object to the value of the global counter. If the cycle count is greater or equal, the object has already been marked either by the same collector or a younger one. Otherwise, the procedure additionally tries to atomically insert its argument into the list of marked objects if it is not part of that list yet.

```
procedure TRACEMARKED( )
    loop
        repeat
            first ← CAS(firstMarked, null, null)
            if first = markedSentinel then
                return
            current ← CAS(firstMarked, first, markedSentinel)
        until current = first
        repeat
            current→TRACE()
            next ← CAS(current→nextMarked, null, null)
            current ← CAS(current→nextMarked, next, null)
        until current = markedSentinel
```

Listing 5.10: Tracing outgoing references of marked objects.

The insertion of an object to the marked list resembles a lock-free push operation of a stack. This list begins with the object referenced by a global variable galled *firstMarked* and ends with a sentinel object referenced by *markedSentinel*. The sentinel allows to distinguish objects that are already part of the marked list from those that are not in which case their respective successor called *nextMarked* is invalid.

The corresponding procedure for tracing and marking outgoing references of marked object is called TRACEMARKED and is shown in Listing 5.10. This procedure is only called by the collectors and is not uncooperative in order to explicitly allow cooperative task switches to mutators. It first tries to get exclusive ownership over the linked list of marked objects by atomically exchanging the reference to the first object with the list sentinel. This allows to traverse and trace the objects in the subsequent step without having to synchronise with concurrent collectors. Replacing the list with an empty one is semantically valid as it only contains objects that have been marked but not yet traversed. At the same time, it also solves the ABA problem as there is no need to remove single objects individually.

For tracing outgoing references of an object, we implemented a somewhat unusual approach. Instead of equipping objects with the required meta data indicating which fields are references and must be marked, the compiler implements an abstract method called TRACE for each object. The generated code just calls the MARK procedure for each reference field or array of references contained in the object. However, how references are identified exactly is an implementation detail of the compiler and it is irrelevant for the garbage collector as long as it is guaranteed that the MARK procedure is called for each outgoing reference. Our design using an abstract method does not pollute the implementation of the garbage collector with these low-level details and allows it manage any other resource just as well.

Finally, the actual garbage collection is shown in Listing 5.11 on the next page. It can be called by several activities concurrently and performs the following steps:

1. Each garbage collection first increments the global cycle count called *cycleCount* and remembers its previous value.

2. A global counter called *oldestCycle* stores the cycle count of the oldest still running collector. If a collector as a higher number than this global counter, it means that there are other collectors which have not yet marked the complete object graph. In this case all younger collectors help the oldest one to trace and mark the object graph completely before it can sweep unreachable objects. Since the mark phase uses always the most current value of the global cycle count, all helping collectors do probably not have to do a lot of marking if they become the oldest one.

3. At this step, the current collector is the oldest one and tries to get exclusive ownership of the watched list which contains all managed objects. This operation needs an atomic compare-and-swap operation as this list is accessed concurrently by mutators which potentially create and insert new objects in parallel. As before, the atomic emptying of the complete watched list also solves the ABA problem.

```
procedure COLLECT( )
    cycle ← INCREMENT(cycleCount)                          ▷ (1)
    while CAS(oldestCycle, 0, 0) ≠ cycle do                ▷ (2)
        TRACEMARKED()
    repeat                                                 ▷ (3)
        first ← CAS(firstWatched, null, null)
        current ← CAS(firstWatched, first, watchedSentinel)
    until current = first
    MARKGLOBALREFERENCES()                                 ▷ (4)
    MARKLOCALREFERENCES(first)
    TRACEMARKED()
    while there are mutators assigning do                  ▷ (5)
        TRACEMARKED()
    while there are collectors tracing do                  ▷ (6)
        TRACEMARKED()
    INCREMENT(oldestCycle)                                 ▷ (7)
    SWEEP(first, cycle)                                    ▷ (8)
```

Listing 5.11: The garbage collection cycle.

4. First, all global reference variables are marked using a call to the MARKGLOBALREFERENCES procedure. Regarding Active Oberon, the global root set consists of the variables associated with all currently loaded modules. The procedure just calls the MARK procedure for all instances of a global reference variable.

   In order to mark local references, the MARKLOCALREFERENCES procedure traverses the exclusively owned watched list and calls MARK whenever the local reference counter of an object is not zero. Subsequently, a single call to the TRACEMARKED procedure marks and traces the complete reachable object graph.

5. At this point, the complete root set and all reachable objects have been marked but concurrent mutators might have already modified the object graph. If there are concurrent assignments, the corresponding write barriers make sure that the respective objects are marked using the MARK procedure. The loop in this step ensures that all marked objects are also traced for outgoing pointers as long as there are other processors executing assignments. As a result, all reference variables modified before or after this step are guaranteed to be marked.

6. This step helps younger activities to finish marking the object graph and effectively synchronises concurrent collectors with each other. This is important for the oldest collector which is about to sweep unreachable objects which otherwise might still be marked and traced by younger collectors helping the oldest one.

7. At this point, all required objects have been definitely marked. By incrementing the lowest cycle count, the next collector is able to proceed which in turn will increment the counter as well.

8. A call to the SWEEP procedure finally deallocates all unreachable objects in the private watched list. The corresponding procedure traverses the linked list and removes all objects with a cycle count not greater than the one remembered in the first step. The removed objects are deallocated and the remaining list of still reachable objects is pushed back to the global list of watched objects. As the collector operates on a private watched list, all objects allocated after the exclusive acquisition can only be deallocated by younger collectors.

Strictly speaking, the algorithm described above is not lock-free among concurrent collectors because they have to wait for each other. If one of them fails to make progress, all younger ones will be blocked forever. A simple solution is to forbid concurrent collectors in which case steps 2 and 7 can be elided. If a collector aborts during the marking phase in this version, it does not affect any subsequent collectors except that the objects in its private watched list can no longer be reclaimed.

Although our approach is not lock-free with respect to collectors among themselves, it still allows to mark and sweep the object graph in a lock-free manner in parallel to concurrent mutators. It relies only on lock-free data structures which do not have any intrusive impact on concurrent mutators except for negligible starvation issues. Apart from scheduling, any activity that does neither mark objects nor trigger a garbage collection by itself, is therefore guaranteed to be never affected by concurrent garbage collections. This class of activities typically include device drivers which profit from this important and practical real-time property.

In summary, the garbage collector presented in this section improves upon existing solutions for the Active Oberon programming language in several areas. First of all, our mark and sweep garbage collector is not bundled with the underlying memory manager and is designed to be an optional and non-integral process of the system just like any other activity. Its mark phase is precise because the compiler provides all necessary meta data in the form of write barriers and abstract methods for tracing objects. Furthermore, the garbage collector applies only lock-free data structures which allows to completely detach mutators from collectors. As a consequence, garbage collections can be performed concurrently and incrementally and therefore have a non-intrusive impact by design. In addition, lock-freedom also allows reentrancy and implies that garbage collectors can be executed in parallel in which case they help each other to distribute the load of marking the object graph. In retrospect, all of these features have basically been enabled thanks to our novel approach of combining lock-free programming with cooperative multitasking.

6 Case Study: Software Portability

This chapter discusses all actions that had to be taken in order to achieve a high portability of the runtime system. This does not only apply to the design of its source code but also includes the implementation of the underlying programming language as well as the abstractions required by the corresponding development tools.

6.1 Introduction

One of the main purposes of an operating system is to provide a generic runtime environment for application software. For this reason, an operating system often incorporates a powerful abstraction layer for interfacing the underlying hardware components in a portable manner. By limiting itself to this high-level application programming interface, carefully designed software does only require a recompilation in order to retarget a different hardware architecture. In an ideal world, this form of software portability should also be applicable to the operating system itself.

Due to the low-level nature of some components of operating systems however, this goal can seldom be accomplished to the full extent. Depending on the architecture, there are many hardware components that require special-purpose code for accessing their functionality. Examples include machine-dependent facilities provided by the central processing unit and other integrated controllers for handling interrupts, managing virtual memory, or interfacing peripheral devices. The corresponding source code for controlling these low-level facilities is typically written in assembly language because it usually requires special machine instructions. This is aggravated by the fact that hardware manufactures often specify proprietary and non-uniform protocols for accessing the respective hardware.

One of the main goals of this thesis was to design a multiprocessor runtime system that minimises the need for machine-dependent code. We achieved this objective by following three fundamental design guidelines:

Prefer Alternatives: Machine-independent alternatives should be preferred to non-portable and hardware-specific solutions wherever possible. Interrupt-driven preemption for instance was abandoned in favour of cooperative multitasking which can be implemented completely in software. Other examples include the runtime checks for error conditions such as null pointer exceptions and stack overflows which are typically implemented using virtual memory.

Unify Runtime Environments: The requirements of runtime environments must be unified to the lowest common denominator of hardware features. The implementation of the runtime system may therefore only rely on hardware facilities which are uniformly provided by each and every hardware architecture. Examples of facilities which are not supported universally include the support for interrupt handling or virtual memory management. If a machine supports such features, it must be set up beforehand in such a way that it can mimic a machine that does not provide them.

Use Compiler Abstractions: Dependencies on the instruction set architecture shall be abstracted by the compiler wherever possible. The implementation of task switches in the runtime system for instance requires access to special-purpose registers like the stack pointer. It also requires a uniform calling convention across different hardware architectures. The corresponding facilities are provided by the compiler in the form of a generic and portable programming interface.

By design, the implementation of our runtime system does only require hardware resources which are also available to ordinary applications running on top of an operating system. As a consequence, we are able to bundle the runtime system as an additional library linked into the binary executable file of an application without requiring special hardware support.

Concerning the kernel, the task of standardising the runtime environment is inherently non-portable but can be completely arranged by an initial bootloader which is typically written in assembly language anyway. Regarding the AMD64 architecture for example, our BIOS bootloader not only loads the kernel image from disk to memory, but also switches the processor state from 16-bit mode to 64-bit mode and identity maps all of the main memory.

Regarding the runtime environment established so far, additional hardware support for facilities like interrupt handling or multiple processors is completely optional. If its implementation is provided, it may differ from one hardware architecture to another or depend on the underlying operating system. The corresponding programming interface consists of a thin layer of abstractions provided by small and replaceable modules. The same applies to substitutable functionality provided by the runtime environment like allocating memory or accessing peripheral devices.

The following sections describe the application of the guidelines in a variety of contexts concerning the development of the runtime system. The key to achieve a high portability was to abstract the underlying execution environment in terms of hardware as well as software.

6.2 Cross Compilation

Source code is usually considered portable when it can be made executable on different hardware architectures or runtime environments without much effort. Ideally, the source code must only be recompiled targeting the new execution environment without having to modify it at all. This implies that the compiler satisfies two different preconditions:

1. The compiler must provide programmers with an abstraction layer for all hardware-dependent issues which may cause a program to be implicitly restricted to one specific platform. Examples include predefined machine peculiarities like the intrinsic bit-width of machine words, alignment constraints, endianness specifications, or calling conventions. Most of these issues can be abstracted by a proper definition of a generic type system.

2. The compiler must be able to translate the same source code for different hardware architectures. It must therefore provide a powerful abstraction of the target instruction set architecture which allows to handle all machine-dependent compilation stages like register allocation and code generation.

In accordance to our goal of implementing a portable runtime system, we naturally wanted the source code of the compiler itself to be portable. By design, the resulting compiler is able to translate source code written in Active Oberon into machine code for any target it supports, regardless of the host machine the compiler is running on. Compilers with this ability are generally called cross compilers but we find the mere necessity for this term unfortunate since all compilers should ideally be able to cross compile. The lack of this capability implies nothing but a compiler that is not portable.

Concerning the original language definition of Active Oberon [Rea04], we had to complement its already powerful feature set with several abstractions in order to allow programmers to write more portable code. Appendix C lists all modifications and enhancements and describes them comprehensively. The most important extension was the introduction of a memory model for lock-free programming as described in Section 2.2.2. This includes the definition of a new built-in procedure implementing an atomic compare-and-swap operation on arbitrary basic or reference types. Other extensions include several new distinct integer types which allow to properly represent addresses, length of arrays, and machine words on architectures with different bit-widths. All of these additions allowed us to write the runtime system in a completely portable manner, targeting a variety of hardware architectures ranging from 8-bit to 64-bit machines.

The compiler is able to target a variety of hardware architectures by utilising a powerful intermediate code representation of the translated source code [Fri11]. This representation consists of a small instruction set and a well-defined programming model for an abstract machine which is able to conglomerate all differences of the supported hardware architectures. The compiler first translates the source code into this representation and uses it in a second step to generate the actual machine code.

This design can be found in many modern compilers and offers several advantages [Muc97]. The deliberately low-level nature of the intermediate code allows a direct translation to machine code which is much simpler than the actual translation from source code to intermediate code. Furthermore, an intermediate code interpreter which emulates an execution environment based on the abstract machine allows to verify the correctness of the translated code without having to rely on the generation and execution of the actual machine code.

The instruction set of the intermediate code representation is listed in Table 6.1 on the following page. The underlying abstract machine operates on uniquely named sections, each storing a sequence of instructions for representing executable code or global data. The corresponding programming model includes a processor that executes instructions stored in code sections and includes a set of typed registers and a stack for storing local variables and parameters.

Our implementation improves on a similar intermediate code representation of the Active Oberon compiler by Reali by supplementing it with portable type definitions [Rea03]. The fact that all of the hardware architectures targeted by his compiler had the same bit-width led to a lot of implicit assumptions in his definition of the intermediate code language which had to be fixed accordingly [Neg06].

6.3 Generic Object Files

Having implemented a cross compiler that is able to generate machine code for a variety of architectures, it was also desirable to have a common file format to store the generated code. Files storing the binary representation of machine code and global data are usually called object files. In addition, they most often contain meta data like symbolic links which refer to code and data stored in the same or other object files. Two of the most popular object file formats are the portable executable file format (PE) used in Windows systems [Mic10] and the executable and linking format (ELF) used by Unix-like systems [Too95].

Category	Instruction	Operation
Memory Layout	`data`	Datum Definition
	`reserve`	Space Reservation
Special Purpose	`nop`	No Operation
	`asm`	Inline Assembly
Data Management	`mov`	Datum Copy
	`conv`	Datum Conversion
	`copy`	Data Copy
	`fill`	Data Initialization
	`cas`	Atomic Compare-And-Swap
Arithmetic	`neg`	Negation
	`add`	Addition
	`sub`	Subtraction
	`mul`	Multiplication
	`div`	Division
	`mod`	Modulo
Logic	`not`	Logical NOT
	`and`	Logical AND
	`or`	Logical OR
	`xor`	Logical Exclusive OR
	`lsh`	Left Shift
	`rsh`	Right Shift
Procedure Call	`pop`	Pop from Stack
	`push`	Push onto Stack
	`call`	Procedure Call
	`return`	Return from Procedure
Branching	`br`	Unconditional Branch
	`breq`	Branch if Equal
	`brne`	Branch if Not Equal
	`brlt`	Branch if Less Than
	`brge`	Branch if Greater Than or Equal

Table 6.1: Instruction set of the intermediate code representation.

Regarding Oberon and its predecessors, compiler writers implementing these programming languages always defined their very own object file format [WG92]. The original Active Oberon compiler by Reali for instance uses an object file format which is tailored to the specifics of this particular programming language [RG06]. Years of experience maintaining and extending this file format revealed that its design is based on many implicit and unnecessary machine dependencies. The main issues with this object file format are summarised below:

- A single object file stores the binary representation of one module and contains all symbolic links to imported procedures and variables. All references to the same procedure or variable are stored in a linked list embodied in the binary code itself. During statically linking and dynamically loading the module at runtime, the corresponding bit pattern storing the offset of the next occurrence of the symbol gets replaced by its actual address.

  While this clever design is space-saving, it also assumes that the bit pattern of each instruction referencing a symbol has always the same format. This precondition was sufficient for the x86 and ARM hardware architectures supported by the original compiler [Adv13, ARM05]. However, there exist several other instruction set architectures that use not one but several different encoding formats for instructions referencing an address. In some cases, the corresponding bit pattern is even discontiguously spread within the instruction format [Atm05].

  Even if the issues about the bit pattern are ignored, the instruction format still dictates how the address has to be patched regarding the endianness and alignment constraints of the target machine. Additionally, there are instructions like branch operations which require the target address to be patched relatively to the address of the instruction rather than absolutely. As a result, the linker and loader have to be aware of the instruction encoding and must therefore be customised anew for each hardware architecture.

- In addition to the binary representation of code and data, object files most often also contain meta data about their contents. In the simplest case, this data just consists of information about the interconnections of code and data stored in different object files as described above.

  In the case of Active Oberon for example, the original object file format contains additional meta data describing exported symbols, executable commands, type descriptors, exception tables, and the locations of outgoing references for precise garbage collection. Other object files such as the PE and ELF file formats contain similar information or a subset thereof [WG92, Too95, Mic10].

  However, all of these object files have in common that the binary data as well as all of the contained meta data are structurally on the same level. This basically means that object file format specifications do not only describe how the binary data is represented but also define the layout and format of meta data. Consequently, any modification on the set of information stored as meta data stringently requires to change the specification of the object file format as well. The corresponding modifications naturally affect all development tools that generate or process object files.

  In the case of Active Oberon for instance, the set of affected development tools include the compiler, the static linker, the dynamic loader, as well as the runtime system. Each time the object file format was modified in order to contain different or additional meta data, all of these components had to be changed correspondingly.

  The static linker posed an additional problem because it tries to emulate the operations of the dynamic loader. It combines object files into a binary image on the host machine that looks exactly like the memory layout of the target machine after dynamically loading object files. This emulation is achieved by executing the same source code as the loader but using a fake heap which directly maps to the binary image. This idea fails completely during cross compiling for architectures that have a differing address size or byte order.

Dissatisfied with the existing solution, we designed a new object file format called generic object file. The main goal of generic object files is to be completely machine-independent and to minimise the amount of meta data in order to resolve all of the issues raised above. As a result, our object file format is not bound to Active Oberon any more as it can contain binary code and meta data for any compiled program. In addition, generic object files contain all the information necessary in order to patch the addresses of symbolic links in a portable manner which allows to unify static and dynamic loading using the very same source code [FN11]. We achieved this result by adhering to the following design guidelines:

Minimise Format: Generic object files contain only that information which is strictly necessary for the loader and linker to arrange the binary contents in memory while resolving the symbolic links.

Object files consist of binary sections which describe contiguous sequences of binary code or data. A section is the main entity of an object file and allows to represent a single global variable, constant string, procedure, or any other language construct that occupies contiguous memory. Each section has therefore a unique name and carries additional meta data that specifies how its binary content has to be arranged in memory.

For example, the type of a section describes the purpose of its binary content which is either executable code or global data. This distinction is necessary for supporting Harvard architectures where sections of different types are arranged in different address spaces. In addition, a section contains a list of so-called fix-ups which reference other sections by name and specify how and where the binary data has to be patched in order to store the address of the referenced section.

For informational purposes, the complete definition of the generic object file format is shown in Figure 6.1 on the next page. It is given in extended Backus–Naur form since generic object files can also be represented textually which demonstrates their simplicity, portability, and maintainability [FN11].

ObjectFile	=	{*Section*}.
Section	=	*Type Name Unit Placement Fixups Bits* {*Fixup*} {*Octet*}.
Type	=	`code` | `initcode` | `bodycode` | `data` | `const`.
Name	=	string.
Unit	=	integer.
Placement	=	`aligned` *Alignment* | `fixed` *Address*.
Alignment	=	*Unit*.
Address	=	*Unit*.
Fixups	=	integer.
Bits	=	integer.
Fixup	=	*Name Patches* { *Patch* }.
Patches	=	integer.
Patch	=	*Mode Displacement Scale Patterns* {*Pattern*} *Offsets* {*Offset*}.
Mode	=	`abs` | `rel`.
Displacement	=	*Unit*.
Scale	=	integer.
Patterns	=	integer.
Pattern	=	*BitOffset Bits*.
BitOffset	=	integer.
Offsets	=	integer.
Offset	=	*Unit*.
Octet	=	hexadecimal-digit hexadecimal-digit.

Figure 6.1: Generic object file format expressed in EBNF.

A similar approach was pursued by Fraser and Hanson in order to achieve a machine-independent linker [FH82]. But instead of providing the static binary contents together with information about how to patch resolved symbols, their linker is in fact an interpreter which evaluates expressions stored in textual object files in order to compute the value of each individual word while linking. However, besides the substantial performance degradation, this approach does not take machine-dependent issues like word size and endianness into consideration.

Embed Meta Data: Any other meta data emitted by the compiler describes the compiled program and is therefore completely irrelevant for linking and loading object files. As this kind of information is only required by the runtime system, it can be directly represented by the binary data embedded in additional sections.

This approach allows to circumvent encoding meta data in a proprietary format while compiling in order to process this information and building corresponding data structures while linking or loading. Instead, the compiler directly generates the data structures as expected by the runtime system on the target machine and stores their contents in corresponding data sections. Cross-references within these global data structures are established using the usual fix-up mechanism provided by the object file format.

In the case of Active Oberon, our new object file format turned out to be generic enough to represent all the necessary meta data like module descriptors, type descriptors, and command interfaces. This even includes exception tables and the locations of outgoing references for precise garbage collection which assures us that this approach is feasible for other compiled languages as well.

As a result, any modification of the set of meta data never required us to change the file format specification nor the static linker or dynamic loader. This increased the maintainability and portability of all development tools and the runtime system substantially.

Another kind of meta data stored in the original object file formats for Oberon and Active Oberon were fine-grained object fingerprints [Cre94]. Fingerprints encode information about the public interface of a module in order to detect interface inconsistencies during compiling and loading a module. As fingerprints are basically hash values of fixed size, it is not guaranteed that two distinct entities always have different fingerprints. The generic object format allows to represent fingerprints as variable sized data sections which encode the public interface unambiguously. The sole purpose of generating these otherwise unused sections is to allow the loader to detect mismatching fingerprints by comparing the binary contents of duplicated data sections.

Another proof of the generality of the new object file format is the fact that it is also able to handle the meta data required to represent completely different object file formats targeting platforms like Windows and Linux. This is accomplished by introducing two or more additional sections that mimic the binary contents of headers and footers of other file formats. Regarding the actual binary file generated by the linker, the binary code and data generated by the compiler are physically encompassed by the contents of these format-specific sections. In order to be able to load the generated binary file on a different runtime environment, only these sections have to be replaced while linking because they completely capture all format-specific information. As a result, the linker processes these sections as usual generating a plain binary file as before and does therefore not have to know anything about the actual object file format of the target platform.

6.4 Runtime System Structure

The runtime system consists of several relatively small and generic Active Oberon modules. Table 6.2 on the facing page lists all modules responsible for lock-free scheduling and memory management. It also references the corresponding sections of this thesis that describe the abstract implementation of each module. The references into Appendix B on the other hand lead to the concrete module interfaces.

Module	Description	See Sections
Queues	Lock-Free Queues	3.5 (B.7)
Activities	Basic Scheduling	4.2 (B.1)
ExclusiveBlocks	Mutual Exclusion	4.3 (B.2)
Processors	Multiprocessing	4.4 (B.6)
Interrupts	Interrupt Handling	4.5 (B.5)
Heaps	Heap Management	5.1 (B.4)
GarbageCollector	Garbage Collection	5.3 (B.3)

Table 6.2: Modules with lock-free algorithms presented in this thesis.

The actual module structure of the runtime system is shown in Figure 6.2 on the next page. It depicts the generic layout and imports of the modules responsible for the complete runtime support for the Active Oberon language. Two of these modules are shaded grey in order to indicate that their implementation depends on the actual target architecture or runtime environment. The runtime support can be decomposed into three different layers where the bottom layer consists of the following four modules responsible for scheduling and multiprocessing as described in Chapter 4:

Processors: This module provides the runtime support for managing multiple processors. As described in Section 4.4, we provide several implementations of this module where all of them share the same interface. If the runtime system is used as an application library, there are two implementations targeted at Windows and Linux based systems which create threads for representing processors. In the native case targeting x86 and AMD64 machines, we provide an implementation which uses the advanced programmable interrupt controller (APIC) in order to handle multiple processors.

Queues: Based on processor-local storage partially enabled by the previous module, this module provides the implementation of lock-free queues as described in Section 3.5.

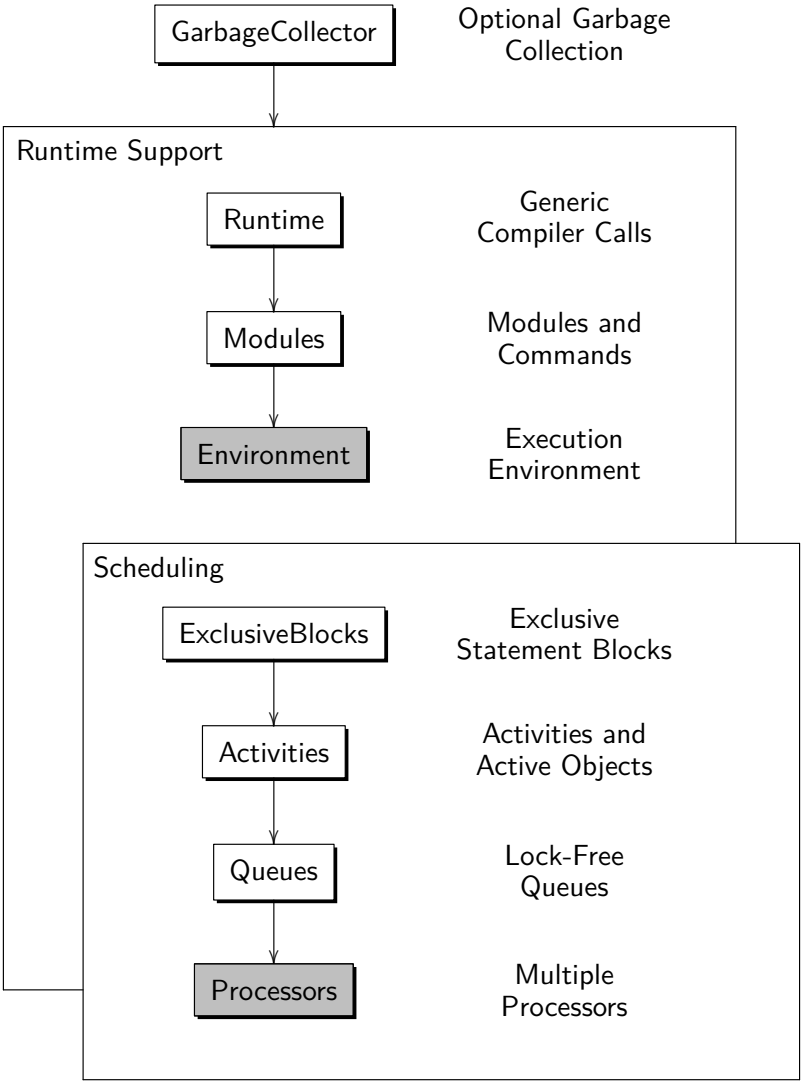

Figure 6.2: Generic module structure of the runtime system.

Activities: This module uses lock-free queues in order to represent activities which are associated with active objects and provides a framework for the implementation of synchronisation primitives.

ExclusiveBlocks: This module makes use of the scheduler framework provided by the previous module in order to implement object monitors which are accessible in Active Oberon by the notion of exclusive statement blocks.

The next layer contains the remaining runtime support which is required for implementing the Active Oberon programming language:

Environment: This module contains a few wrapper procedures for the required functionality provided by the actual execution environment. This includes support for basic input and output as well as allocating and deallocating memory blocks.

Modules: This module is responsible for representing modules and their public interfaces as well as loading the modules in the correct order as predefined by their import hierarchy.

Runtime: Based on all previous modules, this module subsumes all remaining runtime support for various built-in procedures provided by the compiler.

The topmost layer consists of a single module which is not stringently required in order to implement Active Oberon and can therefore be excluded:

GarbageCollector: This module provides the complete functionality for automatic memory management in the form of garbage collection as described in Section 5.3.

If the runtime system is used as a kernel for standalone operating systems, there is basically no underlying runtime environment. Therefore, none of the functionality provided by the execution environment can be forwarded and must be implemented from scratch for the corresponding machine.

Targeting an execution environment based on the Basic Input / Output System (BIOS) for example, the corresponding module has to be replaced by a machine-dependent implementation making use of drivers in order to access peripherals devices for input (PS2 keyboards) and output (serial ports and VGA displays). Figure 6.3 on the next page shows the structure of this particular module and its imports. Naturally, most of its imported modules are machine specific and inherently non-portable. However, the following two modules are still generic and completely portable and can therefore be reused in kernels targeting different machines:

Interrupts: This module provides a synchronisation primitive that allows to await the occurrence of interrupts in a high-level fashion as described in Section 4.5. It makes use of a non-portable module responsible for abstracting the underlying central processing unit of the target machine in order to handle low-level interrupts.

Heaps: This module manages heaps for allocating memory blocks from contiguous memory regions as described in Section 5.1. During booting, the BIOS based kernel performs interrupt calls in order to detect unreserved chunks of memory and uses this module to manage allocations therein.

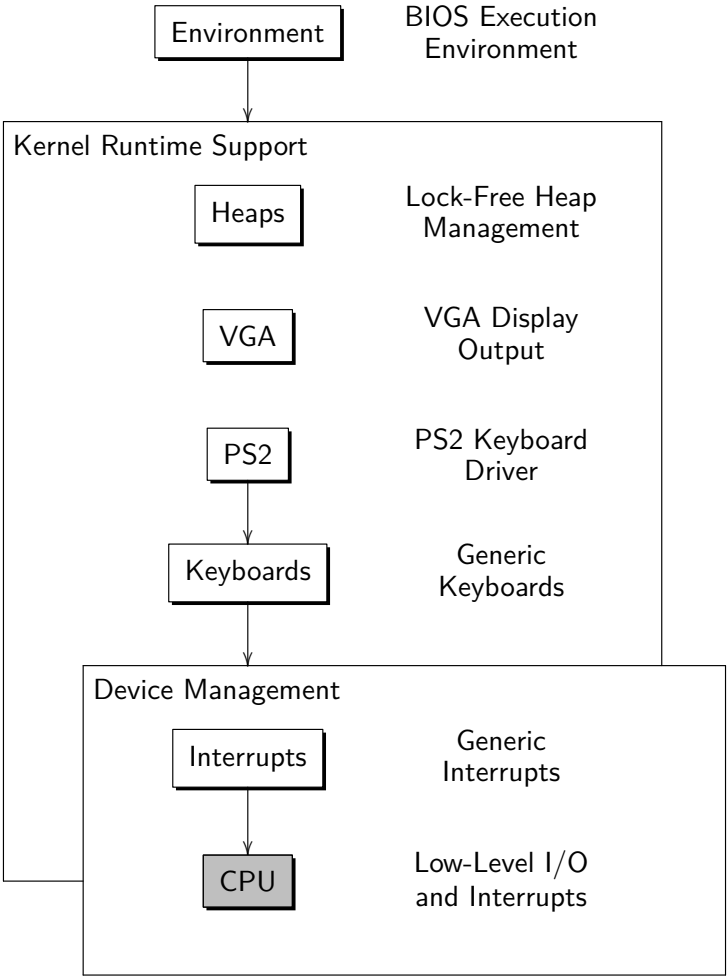

Figure 6.3: Kernel specific module structure of the runtime system targeting an execution environment based on BIOS.

7 Evaluation

This chapter evaluates the performance of several components of our lock-free runtime system and compares the results against related work. Although our system is designed to be portable across hardware architectures, we do not intend to contrast its performance on different machines.

Modern processors and memory hardware seem to apply more and more sophisticated optimisation techniques in order to execute code more efficiently. Some of the most elaborated inventions in this area include for example caching, branch prediction, and out-of-order execution. We do not delve into the details of these features since all of them are typically implemented in hardware and are therefore completely transparent to the programmer [Fog14]. Even though they do speed up the execution of code in general, they often render the performance of processors non-deterministic at the same time.

But even when considering only a single hardware architecture in isolation, the performance of different implementations thereof may vary considerably. In order to be able to minimise all of these effects and to concentrate on the performance of the actual code, we conducted all of our experiments on the very same hardware. This approach makes it difficult to reason about the absolute execution time of our algorithms in general. Instead, our focus is on relating our work to existing solutions if executed under high contention.

We used an AMD64 server machine with 16 GB main memory and two physical AMD Opteron 6272 G34 processors running in 64-bit mode and featuring 16 cores each. For timing our experiments we make use of a built-in high precision hardware timer which has an accuracy of at least 10 MHz. This setup provides a total of 32 logical processors and allows us to the evaluate and compare the performance of our system under increasingly heavy contention.

7.1 Contention Management

Atomic read-modify-write operations like compare-and-swap typically operate on a single memory location. These operations are known to suffer from performance degradation when several contending processors access the same shared memory location concurrently. The compare-and-swap operation for example allows only one processor to succeed while all other contending accesses will fail. Since non-blocking algorithms are usually designed to repeat the whole operation in case of failure, there is a potentially high memory traffic surrounding the shared memory location. The resulting memory congestion degrades the efficiency of cache coherency protocols and may even have an impact on the performance of successful compare-and-swap operations.

Figure 7.1 on the facing page shows the average number of successful compare-and-swap operations while increasing the number of contending logical processors on our server machine. Each participating processor atomically increments a shared counter variable using the algorithm shown in Listing 2.2 on page 26 in a tight loop for the duration of five seconds. The graph shows the value of the counter after all processors have finished which equals to the number of all successful compare-and-swap operations. On this particular hardware, the total number of successful operations is highest for four to nine processors but yields only a marginal speedup with respect to the performance using a single processor. Starting with twenty or more contending processors, the actual throughput of successful compare-and-swap operations decreases significantly. Regarding the full set of logical processors, the actual throughput of successful compare-and-swap operations even drops to about one half of the initial performance. As a result, atomic operations like compare-and-swap must be considered inefficient and expensive under high contention in comparison to standard instructions.

The actual variation of the throughput is shown by the grey box plots in the background of Figure 7.1. It is triggered by the interaction of the varying subset of executing processor cores in each of the twenty measurements per data point. Our scheduler does not implement processor affinities and

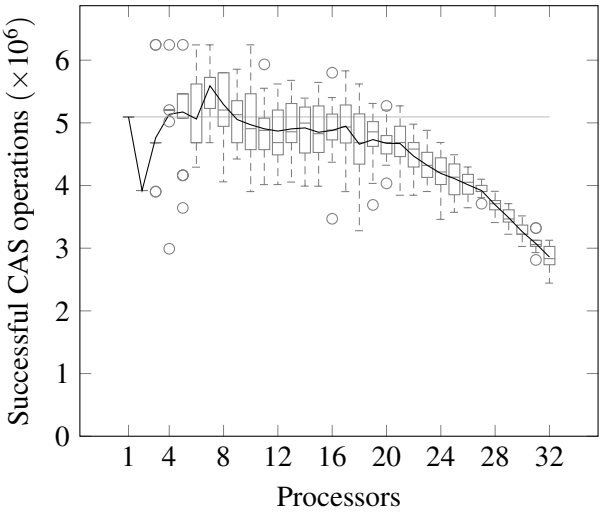

Figure 7.1: Throughput of contending compare-and-swap operations.

basically assigns tasks to those idle processor cores which are awaken first by the hardware upon an interprocessor interrupt. Therefore, the actual subset of running processor cores differs in each run and affects the physical load of the two processor dies. Contending cores running on the same die have different performance characteristics than cores distributed across the physical processors. Therefore, the variation of the throughput correlates with the actual number of all possible combinations to distribute running cores among the dies. Regarding our purposes, this effect can be safely neglected as we are interested only in the overall trends when increasing the amount of logical processors that contend over the same shared variable.

The concept of contention management tries to improve the behaviour of atomic memory operations under high contention. The usual approach is to back off by delaying a contending processor if the atomic operation failed in order to prevent memory devices from congesting. Although this method causes failing processors to potentially wait longer than required, it is capable of increasing the overall throughput of all contending processors.

The same ideas are drawn from other fields with high potential for contention like transactional memory [HM93], locking [And90], or networking [MB76]. Dice et al. provide an excellent overview over contention management algorithms in the context of atomic compare-and-swap operations [DHM13]. They have shown that two of the simplest algorithms called constant backoff and exponential backoff are also the most efficient particularly on the x86 architecture. Both approaches wait for a certain amount of time before repeating a failed compare-and-swap operation. In the case of exponential backoff this delay is not fixed but depends on the number of times the operation has failed since the last successful attempt. A sample implementation of this algorithm is shown in Listing 2.3 on page 32.

We utilise constant backoff instead because it is the simplest contention algorithm but still promises high performance [DHM13]. It basically just performs a busy wait for some platform dependent constant time whenever a compare-and-swap operation fails. The key advantage of this very lightweight implementation is that it does neither require any per thread context information nor depend on the actual non-blocking algorithm.

Figure 7.2 on the next page shows the results of a similar experiment as before which additionally uses constant backoff with varying waiting times. It depicts the total number of successful compare-and-swap operations where failing operations are followed by an increasing number of repetitions of an empty loop. A tight loop consisting of a thousand iterations already helps to improve the vast performance degradation under high congestion considerably. Starting at 10 000 iterations, the overall throughput has improved to such an extent that there is always some speedup with respect to the single processor case. Of course, this number is hardware-dependent but it seems to be a good indicator for other machines as well. Tuning the number of iterations to 10^5 or 10^6 renders the throughput for three or more processors close to constant and almost independent from the actual contention. In both cases there is also hardly any variation any more since the relative measurement error drops down to less than one percent. All of the following experiments on our server machine use a constant backoff along the lines of these results in order to minimise performance penalties under high contention.

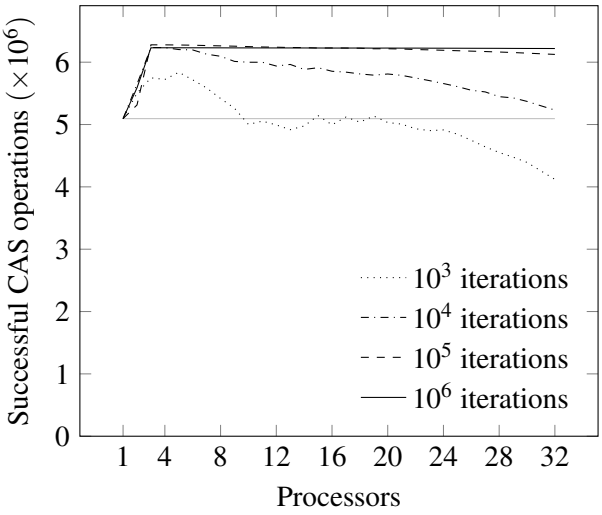

Figure 7.2: Throughput of contending compare-and-swap operations with different constant backoff.

Especially on this particular machine, we also observed that a naive implementation of processor-local storage using arrays often caused false sharing. False sharing describes situations when two or more processors share the same cache line even though they intentionally read and write distinct memory locations therein [TLH94]. As a consequence, any modification of a processor-local variable invalidates the corresponding cache lines of other processors. This results in a cache miss and causes a significant performance degradation each time one of these processors reads its own processor-local variable.

A simple but very effective solution to this problem is to align the elements of the global array to the actual size of the cache lines. This may introduce some spatial overhead but basically ensures that processor-local storage is never shared amongst the caches of two different processors. The same technique may also be beneficial for variables that are modified using atomic compare-and-swap operations in lock-free algorithms.

7.2 Performance Measurements

The main goals of our runtime system were to increase its correctness and reliability by abolishing any kind of lock-based synchronisation while striving at the same time for an uncompromising portability and simplicity of its source code. But nevertheless, any system designed to run on shared-memory processors should naturally also make the most out of the available parallel processing power. In this section, we want to demonstrate that our approach has indeed proven itself in practice and is sometimes even able to compete to some extent with the performance of highly optimised operating systems.

In order to quantitatively compare the performance of various parts of our lock-free runtime system against the corresponding services provided by established operating systems like Windows and Linux based systems, we assembled a broad set of micro-benchmarks. In comparison to standard benchmarks which are often complete applications that use several system-specific features simultaneously, micro-benchmarks are simple programs that allow to focus on a single feature in isolation in order to understand its performance in the general case. For several reasons however, the results presented in this section should be taken with a grain of salt:

First of all, micro-benchmarks in general measure the performance in worst-case scenarios rather than the actual runtime behaviour of full-blown applications and are therefore not very realistic [BDF92]. In absence of a real-life workload, memory caches in particular are likely to behave completely different and may cause the performance of memory-bound micro-benchmarks to become distorted.

In addition, our runtime system follows a strictly machine-independent design and has therefore a much smaller set of features than other systems. For example, it deliberately abandons virtual memory management and does therefore neither support memory swapping nor heavy-weight processes with separate virtual address spaces. As a result, the micro-benchmarks are only able to cover a small overlap of the functionality provided by the other systems which may differ substantially in their implementation and sophistication.

Each micro-benchmark compares up to four different operating systems running on the same machine. The first one is always the standalone version of our lock-free runtime system called Native hereafter. The second one is an Ubuntu Server operating system based on Linux version 3.13.0 and has GNU libc version 2.19 installed. Where applicable, the benchmarks are also executed on Windows 7 as well as the latest version of the A2 operating system.

Unfortunately, A2 runs in protected mode instead of 64-bit mode and can therefore only access about a quarter of the 16 GB main memory. In addition, it requires to use another set of development tools which is why the actually executed machine code is slightly different.

In order to focus on the actual performance of these four runtime environments, we tried to minimise the differences of the machine code generated for each micro-benchmark. Therefore we developed a retargetable framework that allows us to use the exact same source code as well as the same 64-bit compiler in order to create similar executable files for the different systems. Except where otherwise noted, the generated binary code for the benchmarks differs only with respect to the runtime calls for managing threads and allocating memory. Targeting our own runtime system and A2, we make use of the built-in language facilities implemented by the respective compiler. In the case of Linux, we use the default C and PThread libraries and call functions like `pthread_create` and `malloc` instead. Executable files targeting Windows on the other hand make use of the corresponding functions of the Windows API.

Each of the following micro-benchmarks is repeated 20 times in order to compute the arithmetic mean and relative measurement error. Depending on the actual benchmark, we measure either the wall-clock time using the contention-less high-precision timer or the throughput of programs running for a fixed amount of time. Where the behaviour under different levels of contention is of interest, we created an increasing number of contending threads on an otherwise completely idle machine. In order to unify the hardware-specific behaviour of the two physical processors under increasing workload, we also disabled all respective BIOS features like power management and performance boosting.

Environment	System Call	Time	Error
Native	`Activities.GetCurrent`	5.2ns	0.0%
A2	`Objects.ActiveObject`	12.7ns	0.0%
Linux	`sys_gettid`	1115.3ns	0.1%
Windows	`GetCurrentThreadId`	9.8ns	0.4%

Table 7.1: Duration of minimal system calls.

7.2.1 Minimal System Call

One of the simplest test cases for comparing the performance of operating systems is the evaluation of the actual overhead of calling a system service in the first place. One of the cheapest system calls of Linux is called `sys_gettid` which just returns the unique identification number of the calling thread. On Windows, the same functionality is provided by the `GetCurrentThreadId` function. The corresponding functions of our runtime system and A2 are called `Activities.GetCurrent` and `Objects.ActiveObject` respectively and return the unique descriptor of the currently running activity.

Table 7.1 shows the average overhead of a system call when invoking these functions 10 million times in a row. In Windows and our runtime system the requested information is readily available using a dedicated register and takes the least time. A2 needs to query the descriptor at a location near the very beginning of the current call stack which must be computed first. Linux essentially performs the same operation but has a high overhead compared to the other systems because the system call first jumps from user space to kernel space by issuing a software interrupt. This jump is necessary to get access to internal data structures of the kernel which are otherwise protected by hardware mechanisms from malicious user code. In all other cases including Windows however, this particular system call equals an ordinary function invocation which does not offer this kind of protection. In A2 and our runtime system, all of the provided system calls are in fact statically bound.

7.2.2 Thread Creation Time

A further important performance characteristic of a multitasking operating system is the overhead in terms of the time it takes to create and terminate tasks. In A2 and our runtime system, tasks are called activities and are equivalent to lightweight processes or threads known from other systems. Since activities are associated with active objects, they are automatically created by instantiating an active object. In order to create threads under Windows and Linux on the other hand, we call the library and API functions `pthread_create` and `CreateThread` respectively. All threads just terminate their execution as soon as they are scheduled.

Figure 7.3 on the following page shows for each system the total time it takes to create and wait for the termination of an increasing number of threads. The benchmark and all of the created tasks run on the same core in order to properly estimate the overall overhead per thread. The results clearly show that A2 and especially our runtime system outperform the other systems significantly.

In all four cases, the overhead of creating a thread consists at least of reserving memory for the stack and registering the thread for scheduling. A2, Windows, and Linux additionally append the reserved stack memory to the page table of the virtual memory management. This technique allows to detect and react to stack overflows which trigger a page fault if the accessed memory location on the stack is not mapped. However, this approach also implies that the systems have to reserve several consecutive virtual memory pages per thread. The considerable overhead of Windows and Linux in comparison to A2 suggests that there is some additional work involved.

Our runtime system relies on compiler-inserted checks rather than on virtual memory management and increases a stack if necessary by copying its contents into a larger memory block as described in Section 5.2. As a result, the stack of newly created threads can have a much smaller granularity in comparison to the other systems. This is the reason why our runtime system is the only one that can handle more than one million threads at the same time. A2 and Linux fail to create 100 000 threads at once while Windows struggles with one million.

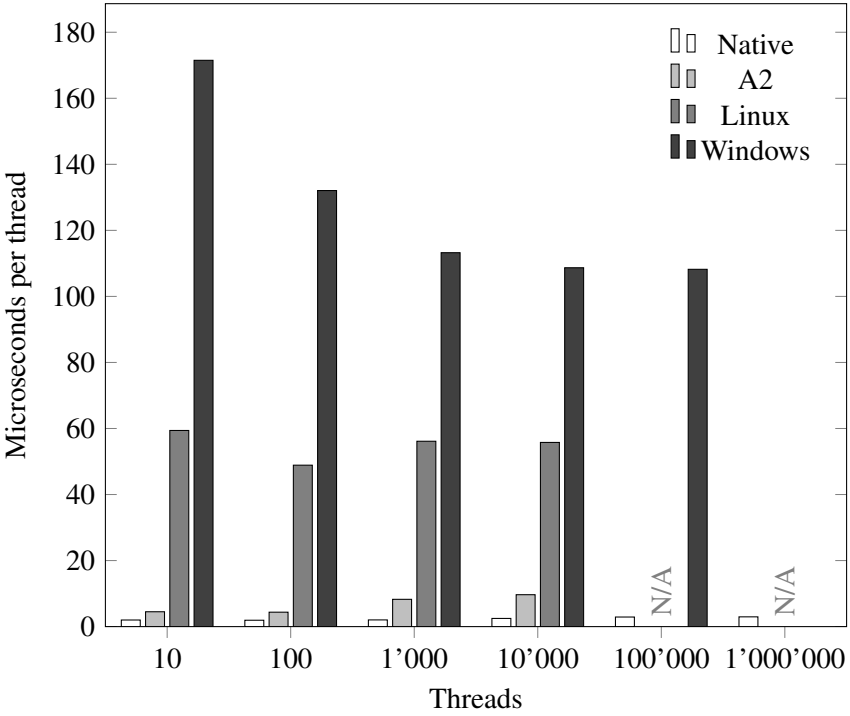

Figure 7.3: Thread creation time.

7.2.3 Context Switch Time

The next micro-benchmark tries to estimate the overhead of a single context switch under high contention. The context switch time in this case describes the time it takes the scheduler to select a new task to run and to give it the control of execution. The corresponding functions to perform an explicit context switch are called `Activities.Switch` in our runtime system, `Objects.Yield` in A2 and `pthread_yield` under Linux. Windows does not offer an API function that guarantees a context switch and is therefore not considered in this benchmark.

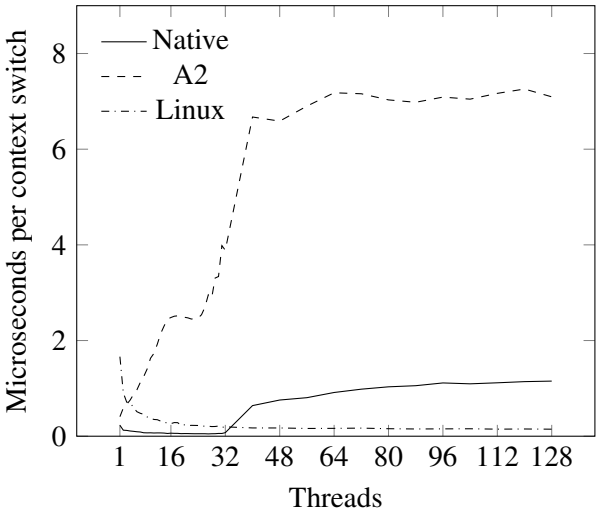

Figure 7.4: Context switch time.

Figure 7.4 shows for each system the average cost of a context switch under increasing contention. We measured the time it takes all threads to perform 10 000 context switches in a tight loop and divided it by the total number of context switches. A2 and our runtime system perform essentially the same operations but A2 protects its ready queue using a global spinlock. Regarding 1 to 32 threads, there is in both cases hardly any context switch necessary because the ready queue is empty most of the time. The increasing time difference however stems from the spinlock which forces contending threads to wait for each other.

From 32 threads onwards, the ready queue starts to get filled and context switches become necessary. The overhead of our runtime system is mainly due to the increasing memory contention on the global ready queue but it still performs several times better than A2. Starting at 64 threads which are permanently task switching, there is at least one item per processor in the ready queue and the overhead per context switch becomes more or less constant in both cases.

The results for Linux on the other hand expose a radically different performance in comparison to A2 and our runtime system. According to the source code of Linux and libc, a call to the `pthread_yield` function eventually triggers the `sched_yield` system call which uses a dedicated ready queue per processor. For performance reasons, these queues are infrequently load balanced and therefore require hardly any synchronisation amongst each other which effectively prevents memory contention. Contrary to its specification however, this optimisation implies that the `pthread_yield` function can therefore not guarantee that another thread which is ready to run actually gets executed. This is why the authors of the Linux kernel strictly forbid users of the `sched_yield` system call to misuse it to synchronise threads with each other.

It is important to keep in mind that this micro-benchmark tests the performance of the system in the absolute worst case where threads perform nothing else but context switches. As soon as there is some additional workload, there is much less memory contention and the switches are bound to perform better in our runtime system. This is not the case for Linux as can be seen by the context switch time for a single thread which is higher than the worst-case overhead of our runtime system.

7.2.4 Matrix Multiplication

Whereas the previous benchmark measured the time of synchronous task switches, the following one tries to estimate the relative cost of asynchronous task switches. This benchmark measures the overall time required to multiply two 1024 by 1024 matrices with several threads as an example of a simple yet non-trivial concurrent program. It distributes the workload to an increasing number of threads where each thread computes distinct rows of the result matrix.

Since this partitioning does not require any synchronisation of the threads, the actual speedup of the multiplication should be close to optimal. However, because each thread executes exactly the same code on each system, the actual difference in time must originate from the overhead of performing task switches as well as creating and terminating the threads.

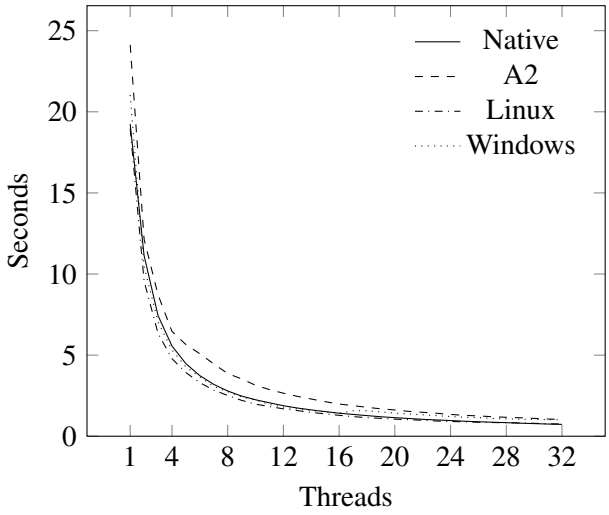

Figure 7.5: Time for multiplying two 1024×1024 matrices concurrently.

Figure 7.5 depicts the overall time for multiplying both matrices using an increasing number of threads on each system. Except for A2, the benchmark yields approximately the same performance on each system. During the computation using a single thread which takes about 19 seconds, Linux performs nearly 1800 asynchronous task switches. A2 interrupts the computation exactly once per millisecond and therefore requires about 24000 asynchronous task switches.

Our runtime system on the other hand performs almost as many synchronous task switches instead. In addition to the actual task switches, the implicitly generated machine code at the end of each nested loop of the matrix multiplication executes a total number of about 1024^3 quantum checks. Compared to a variant of this benchmark that uses an uncooperative block in order to completely prevent implicit cooperative multitasking, the impact on the performance is about 2%. The results shown above indicate however, that the overall overhead of implicit cooperative multitasking is on a par with preemptive scheduling as implemented by Linux and Windows.

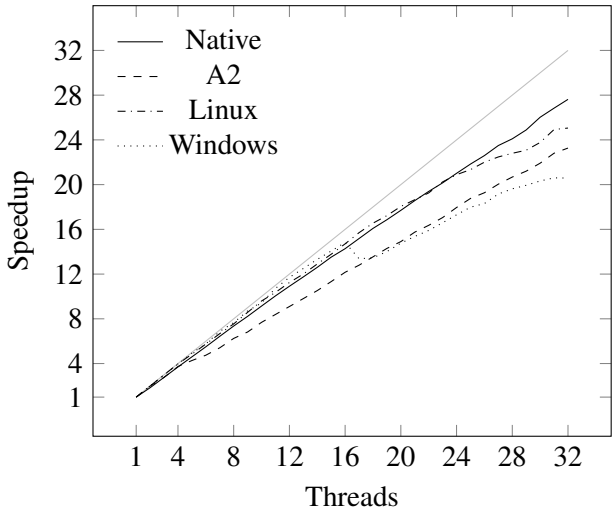

Figure 7.6: Speedup of multiplying two 1024×1024 matrices.

The actual speedup of the concurrent matrix multiplication regarding each system in isolation is shown in Figure 7.6. Under high contention, our runtime system exhibits the highest speedup which is close to linear. The other systems behave similarly but their relative performance drops earlier on when increasing the utilisation of the 32 cores.

7.2.5 Locking

This micro-benchmark was designed to measure the average locking overhead for critical sections if there is none or heavy lock contention. It creates an increasing number of threads that repeatedly enter and exit either the same or their own critical section. The benchmark measures the time it takes all threads to execute 10000 pairs of lock operations in order to compute the average overhead per critical section. In the case of A2 and our runtime system, the benchmark enters critical sections using the built-in object monitor associated with active objects. Under Linux and Windows,

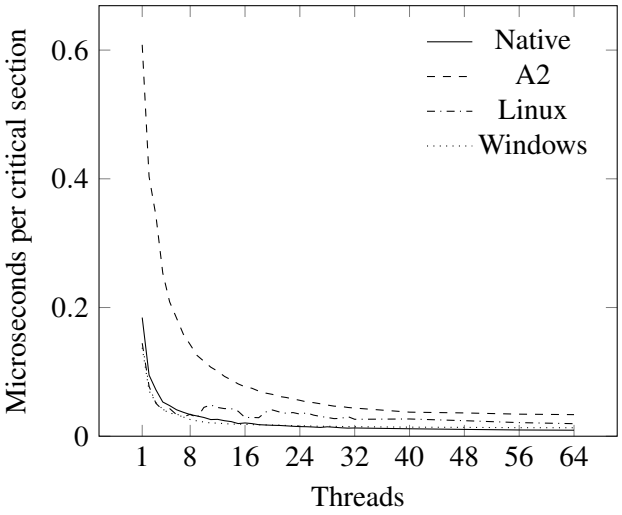

Figure 7.7: Average cost of locking operations with no lock contention.

mutex objects like the `pthread_mutex_t` type from the PThread library and the `CRITICAL_SECTION` provided by the Windows API are used to enter a critical section. These are fast user-space objects since they only require a system call if the mutex is contended.

Figure 7.7 shows the average overhead of entering independent critical sections resulting in no lock contention. As expected, the performance of all systems scales nicely since no contention is involved. A2 reveals a much higher overhead per critical section while all other systems approximately perform the same.

Figure 7.8 on the next page shows the result of increasing the lock contention by entering always the same critical section. A2 reveals an exceptionally high cost under increasing contention which might be caused by the spinlock used to protect the monitor itself from concurrent access while acquiring it. Our runtime system performs much better in this respect but in comparison to Windows and Linux still suffers from some overhead caused by the memory contention.

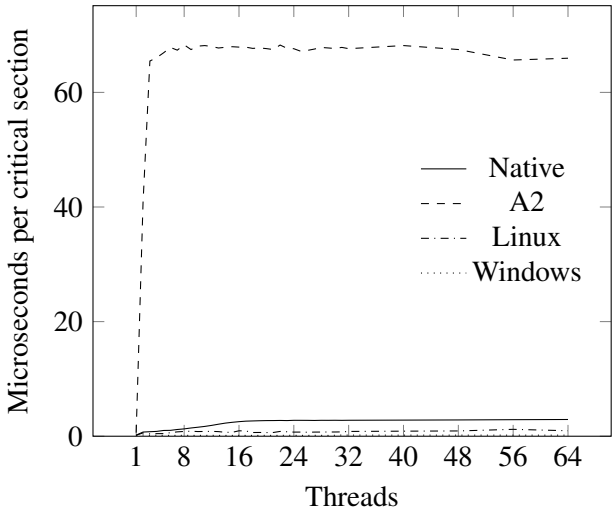

Figure 7.8: Average cost of locking operations with heavy lock contention.

7.2.6 Thread Synchronisation

The following micro-benchmark was designed to evaluate the cost of synchronising several threads with each other under increasing contention. For this purpose, it simulates an increasing number of concurrent runs of independent games. Each game consists of ten players passing a token around in a round robin fashion for a total of 10 000 rounds. Each player is represented by a thread that waits until it receives a token from the previous player. It immediately passes the token on to the next player and waits for it to arrive again in the next round.

In A2 and our runtime system, the wait for the token is implemented using the `AWAIT` statement provided by the Active Oberon programming language. In Linux and Windows on the other hand, the library types `pthread_cond_t` and `CONDITION_VARIABLE` are used instead. In either case, a thread receiving the token will be resumed by another thread which immediately suspends itself again.

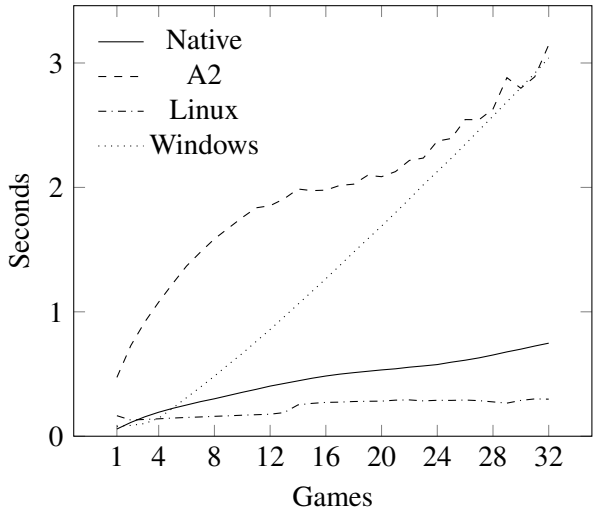

Figure 7.9: Duration of several concurrent games featuring ten players passing a token 10 000 times around.

Figure 7.9 shows the total elapsed time to run up to 32 concurrent games on each of the four systems. Since in each game only one player at a time is running in principle, each game could in theory run on its own processor and the total time should therefore always be about the same.

However, all systems reveal a more or less drastic deviation from the ideal speedup when simulating a lot of games which we attribute to the increased memory contention. The contention is due to awoken players which get enqueued into the ready queue and might continue their execution on a different processor.

As explained above, Linux dedicates a ready queue to each processor and has therefore the smallest overhead. A2 as well as our runtime system use shared ready queues instead which naturally causes more contention. The growing difference in the overall running time of the two systems however is due to the fact that our runtime system does not use a global lock to protect the ready queues from concurrent access.

7.2.7 Memory Allocation

The memory allocation micro-benchmark was designed to assess and compare the performance of the heap managers of all considered systems under high contention. The benchmark creates an increasing number of threads which allocate and deallocate memory blocks in the following intricate way in order to stress the memory managers as much as possible.

Each thread allocates a total number of 100000 memory blocks of random size by performing 10000 pairs of alternating allocation and deallocation operations ten times in a row. This interleaving ensures an evenly distributed load of the two operations independent from the number of concurrent threads. The sizes of the allocated memory are chosen randomly in order to strain data structures like free lists equally. The resulting addresses of the 10000 allocations are stored in a buffer which is thereafter inserted in a global lock-free pool of buffers. Each thread retrieves one buffer from this pool in order to perform the 10000 memory deallocations in reverse order. The usage of these buffers guarantees that memory blocks are never allocated and deallocated by the same thread which places an additional burden on the memory manager.

Benchmarking our native runtime system, we use the built-in `NEW` and `DISPOSE` functions provided by the programming language which map onto the operations of the lock-free heap manager presented in Section 5.1. Unfortunately, this micro-benchmark cannot be executed under A2 since this system relies heavily on the garbage collector and does not allow its users to deallocate memory explicitly. In order to allocate and deallocate memory in Linux, we call the libc functions `malloc` and `free` which are based on ptmalloc2 by Doug Lea and Wolfram Gloger [Glo06]. Under Windows we use the API functions `HeapAlloc` and `HeapFree` instead.

Figure 7.10 on the facing page shows the results of running the benchmarks on all three systems with three different upper limits for the random sizes of the memory blocks. In comparison to the other systems, the results of all measurements clearly indicate that the performance of our heap manager stays the same regardless of the actual block size. The overhead per thread grows more or less linearly in each case and is mainly due to

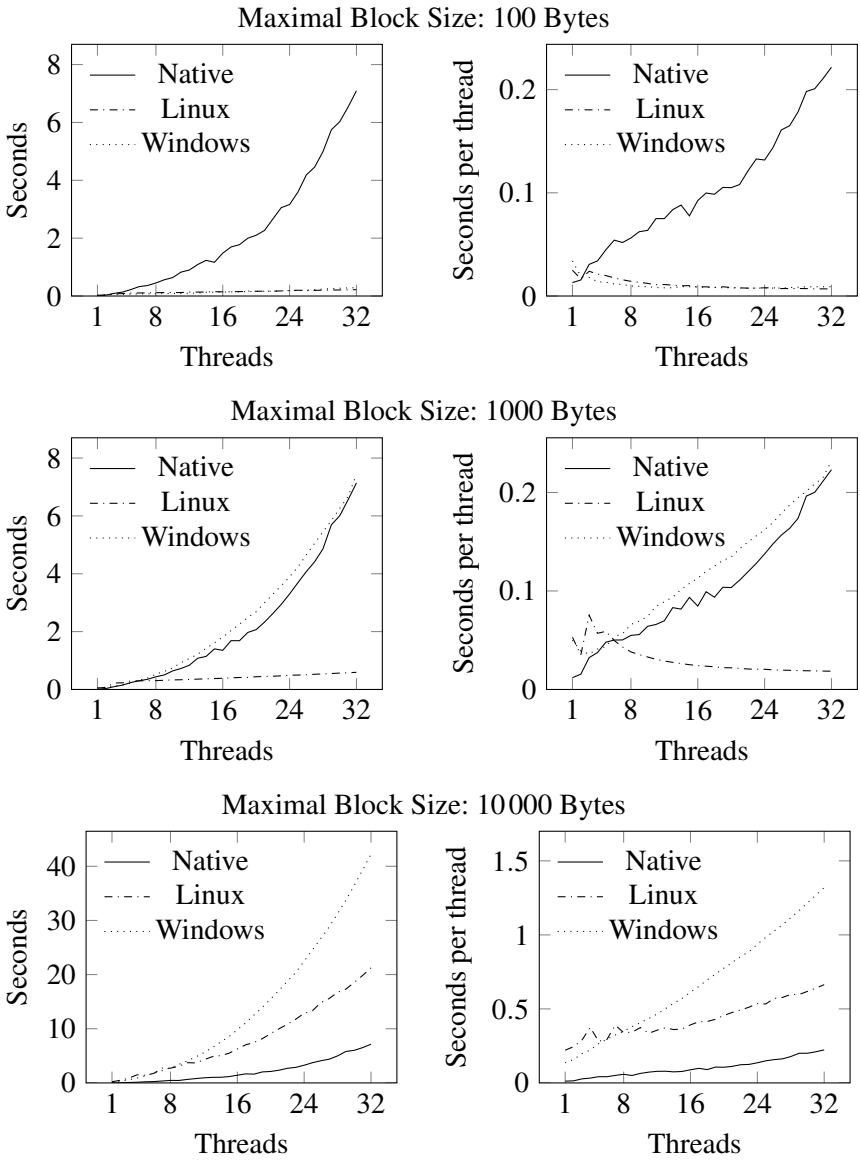

Figure 7.10: Allocating memory with varying maximal block sizes.

the heavy memory contention surrounding the lock-free heap operations. Linux and Windows on the other hand pursue different allocation strategies depending on the size of the memory allocation. For small memory blocks, both systems optimise the synchronisation overhead of concurrent access by allocating memory blocks from preallocated superblocks dedicated to each processor [Glo06, RSI12]. This is demonstrated by the respective measurement results for a maximal block size of 100 bytes indicating no memory contention whatsoever. Increasing the maximal block size reduces the efficiency of using superblocks since they themselves have to be allocated using a different strategy. This is for example shown by the increased overhead of Windows when allocating memory blocks with maximal size of 1000 bytes. The length of superblocks usually equals the page size of the machine which is often exceeded when allocating random sized memory blocks of up to 10 000 bytes. In this case, both systems presumably use locks to protect the heap from concurrent access which is clearly indicated by their exponentially growing run time.

This benchmark shows that our heap manager scales well regarding the size of the requested memory and clearly outperforms implementations based on locking synchronisation. This certainly includes A2 which always allocates memory under mutual exclusion. However, the results also reveal that there is much room for improvement for example by using processor-local memory arenas in order to prevent memory contention altogether.

7.2.8 Memory Management

Automatic memory management in the form of the garbage collector is optional in our runtime system and can be replaced by a manual memory deallocation if desired. The following benchmark tries to compare the performance of programs that either rely on the garbage collector or free their allocated memory by hand. For this purpose, three different exemplary data structures are used to store a total of 2^{22} allocated objects called nodes. The data structures in question are a contiguous array storing references to the nodes, a singly linked list chaining the nodes, and a fully populated binary tree of nodes.

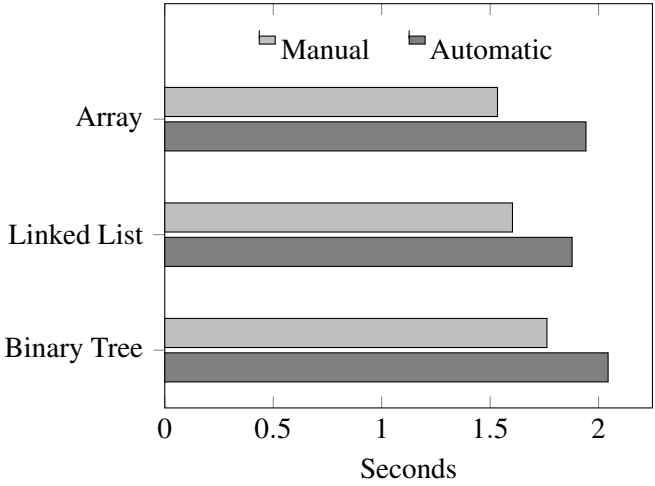

Figure 7.11: Comparison of manual and automatic memory management.

Figure 7.11 shows the total running time of allocating and deallocating the nodes and storing them in one of the three different data structures. For each data structure, the nodes are deallocated either manually or using the garbage collector by invalidating the reference to the root node or the array and performing a complete garbage collection cycle afterwards. The benchmark measures the time for allocating the nodes as well because each allocation includes the insertion of the node into the watched list in the cases where the garbage collector is used.

Regarding the manual deallocation of the array which can be traversed most efficiently out of the three data structures, the overhead of using an automatic memory management corresponds to about 26%. In order to deallocate the nodes stored in the linked list, the manual deallocation must traverse the successor of each node. The binary tree on the other hand is explicitly deallocated using a recursive function. In both cases, the actual traversal of the data structure takes therefore more time than iterating over an array and the overhead introduced by the garbage collector reduces to approximately 17%.

Because this benchmark does not require the garbage collector to mark any objects, the overhead consists only of adding newly created nodes into the linked list of watched objects and traversing them sequentially while sweeping. As described in Section 5.3, the runtime system itself does not rely on the garbage collector and deallocates all of its resources manually.

7.2.9 Garbage Collection Latency

An important characteristic of a runtime system with automatic memory management is the latency caused by the garbage collector. The following micro-benchmark tries to measure the impact of garbage collections on concurrently running threads. For this purpose, the benchmark creates 32 threads in order to keep all cores of the server machine busy for ten minutes. One of these threads repeatedly pauses for a short while and measures the actual wall-clock time it was delayed. All other threads allocate objects of 1000 bytes in a tight loop and store the references to the last 5000 most recent allocations in a buffer. The idea is to continuously trigger the garbage collector with a realistic workload of alive objects and measure its impact on the one thread that does not perform any memory allocations at all.

Windows and Linux being operating systems rather than runtime systems for programming languages do not offer the functionality of automatic memory management. Therefore, we compared the performance of A2 and our runtime system with the Java virtual machine provided by OpenJDK version 7 running on top of Linux. As this Java platform allows its users to select one of several different garbage collectors, we executed the benchmark once for each available option. In order to achieve a runtime environment comparable to the native case, we additionally configured the initial and maximal heap size to compromise all of the 16 GB of main memory where applicable.

Figure 7.12 on the next page shows the percentile distribution of the latencies of all considered runtime systems and their garbage collectors. In 98% of all cases, none of the garbage collectors causes any considerable delay on the measuring thread. Regarding higher percentiles however, the various garbage collectors of the Java virtual machine one after another

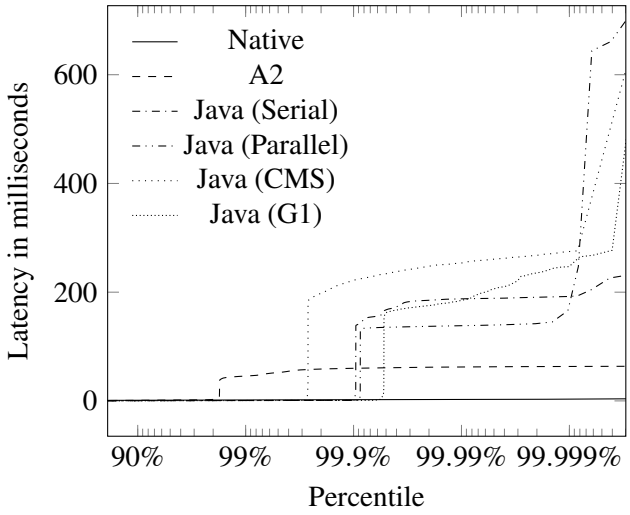

Figure 7.12: Percentile distribution of garbage collectors latencies.

begin to reveal latencies of up to 200 milliseconds and more. This even applies to the concurrent mark sweep collector (CMS) and its successor called G1, both of which are concurrent as well as incremental and designed to decrease garbage collection pauses.

In contrast, the automatic memory management of A2 employs a monolithic stop-the-world garbage collector which causes a maximal latency of about 60 milliseconds. This relatively low duration stays almost constant because it consists only of the time required to mark the object graph which always contains the same amount of objects in this benchmark. The actual sweep phase of the garbage collection does not cause any delay on the measuring thread as it is lazily deferred to the object allocation performed by the other threads.

The garbage collector of our runtime system on the other hand has no intrusive impact whatsoever. This behaviour is to be expected as each thread which detects an out of memory situation executes a garbage collection cycle on its own. Since our garbage collector by design never interferes

with concurrent mutators or independent activities, there cannot be any observable latency apart from scheduling. This would even apply to more realistic settings where garbage collection cycles are usually also performed periodically by dedicated activities.

7.3 Portability and Flexibility

In this last section of the evaluation of our runtime system we try to qualitatively evaluate its portability and flexibility in comparison to other systems.

First of all, the design goals of achieving an uncompromising portability and simplicity of the runtime system are directly reflected by the relative shortness of its source code. Table 7.2 on the facing page for example shows the complexity of the complete runtime system in terms of lines of code in comparison to the equivalent source code of A2. The numbers have been generated using cloc, a program that counts physical lines of source code by removing blank lines and all comments. Although both systems provide a similar set of features regarding the runtime support for Active Oberon, our complete runtime system amounts to about 27% of the source code of the A2 kernel. A notable component is the memory management which requires only 357 physical lines of code for the implementation of the complete heap manager including the garbage collector.

Comparing A2 and our runtime system with other systems is in most cases either not feasible or unfair as they differ significantly in their design and implementation. The scheduler of Linux for example provides a myriad of additional features in direct comparison with our simplistic cooperative scheduler. The corresponding C source code files named core.c, fair.c, and wait.c implement basic scheduling primitives as well as the completely fair scheduler [Mol07] and require almost twenty times more physical lines of source code. The libc memory manager provided in the files malloc.c and arena.c on the other hand needs about ten times more code. In general however, comparisons based on lines of code highly depend on the underlying programming language as well as coding styles and must therefore always be taken with a grain of salt.

Component	Lines	Ratio
Interrupt Handling	299	75%
Memory Management	357	20%
Modules	82	17%
Multiprocessing	215	37%
Runtime Support	174	12%
Scheduler	547	37%
Total	1674	27%

Table 7.2: Size of various components of the runtime system in terms of physical lines of code and its ratio in comparison to A2.

The relative shortness of its source code also automatically increases the maintainability of the system. This asset is boosted by the fact that most of the source code is completely portable and thus a simple recompilation suffices to target a different machine. Only a total of six different functions have to be rewritten in order to support memory allocations, high-level interrupt handling, and multiple processors on a new target. For this reason, the runtime system executes most of the time the same source code and therefore behaves reliably and predictably on each machine it has been ported to. This includes a broad range of platforms like tiny 8-bit AVR microchips with only two kilobytes of SRAM, popular 32-bit ARM-based boards like the Raspberry Pi, as well as powerful AMD64 server machines with 64 cores and 128 GB of main memory.

Its low consumption of resources and its high platform independence even render our runtime system interesting for architectures without shared memory. In multiprocessing systems which use message passing instead, the runtime system can be applied to provide lightweight multitasking on a single processing unit. This is made possible for the most part by using cooperative multitasking instead of preemption which allows the runtime system to be even used on architectures that do not support interrupts or hardware timers.

8 Conclusion

This final chapter concludes this thesis by summarising our work and giving various directions for possible improvements in the future. A comprehensive list of the main contributions of this work can be found in Section 1.4 on page 9.

8.1 Summary

In this dissertation we explored the synergy of lock-free programming and cooperative multitasking in order to create a lock-free and portable runtime system for modern object-oriented programming languages with support for multithreading. Although both techniques have been investigated intensively in isolation, their combination presents a novel and hardly researched approach to lock-free programming. The capability of hindering a non-blocking algorithm from involuntarily losing the control of execution renders several beneficial programming techniques possible. Besides enabling practical progress guarantees for instance, this approach most notably allows non-blocking algorithms to replace thread-local storage with processor-local storage. We showed how this programming technique facilitates lock-free algorithms to be bound only on the static number of processors rather than the dynamic set of threads. The maximal number of processors is always known in advance for any hardware architecture, which allows to implement processor-local storage efficiently and also readily accessible for inspection by any other processor.

One particular lock-free data structure that profits from this programming technique are queues which employ the notion of hazard pointers in order to guard intermediate nodes from unsafe reclamation. We explained how our approach reduces the complexity of identifying hazardous nodes from

linear time in the number of threads to constant time. This enabled us to provide a memory-efficient implementation of a lock-free queue that reuses intermediate nodes if the same elements are enqueued several times. Queue operations do therefore not constantly allocate memory any more, an issue that seems to be taken as granted in the standard literature but has never been addressed before.

We used this improved lock-free queue extensively for the implementation of the cooperative task scheduler because tasks qualify as elements that are potentially enqueued and dequeued a lot during their lifetime. The queues are part of a framework provided by the scheduler to implement various synchronisation primitives in a lock-free manner. We explored simple constructs like mutexes and events but also showed the implementation of a full-featured object monitor. All of these synchronisation primitives make use of the notion of task switch finalisers which are essential to recheck synchronisation conditions in a non-blocking context.

In addition, we also provided a lock-free synchronisation primitive for handling interrupts by awaiting their occurrence in a high-level fashion. The design goal of implementing all components of the runtime system in a lock-free manner has especially paid off in the context of implementing low-level interrupt handlers. Since lock-free algorithms are inherently reentrant, the complete set of data structures and operations of the runtime system are available for use in interrupt handlers without further precaution. Consequently, it is never necessary to disable interrupts which renders the system as responsive as possible.

By replacing preemption with cooperative multitasking, the implementation of the scheduler and the runtime system has basically become machine-independent and its source code is therefore highly portable. In order to port the system to different architectures, only a small set of half a dozen functions have to be adapted. While the runtime system can be used as a kernel providing the essential service routines, its machine independence also allows it to be linked as a library for applications running on top of existing operating systems. The cooperativeness of the tasks created by these applications are enforced by the compiler which supplements the generated machine code with implicit cooperative task switches.

In order to close the circle of the non-blocking runtime support for object-oriented programming languages we also designed and implemented a lock-free memory management. It consists of a lock-free memory allocator that is capable of managing the complete main memory using one or more distinct heaps represented as buddy systems. The memory associated with the call stacks of activities is also represented using ordinary heap blocks. This representation is enabled by runtime checks provided by the compiler which render virtual memory management unnecessary in order to detect stack overflows. Last but not least, we also devised a completely optional garbage collector based on a lock-free mark and sweep algorithm which naturally implies concurrent and incremental garbage collection cycles.

Our lock-free runtime system is based on the kernel of the A2 operating system and designed to serve as an in-place replacement. The source code of the runtime system is much smaller and therefore more maintainable in comparison to the A2 kernel which is based on blocking synchronisation. The evaluation also revealed that our runtime system outperforms A2 in various aspects because of its non-blocking nature. Regarding other systems like Windows and the Linux kernel, our work demonstrated comparable performance in several cases although it provides a much different feature set and abandons optimisations in favour of clarity and simplicity.

However, all of our measurements have shown that the naive perception that lock-free algorithms must by design be more efficient than their blocking counterparts is misguided at best. With an increasing level of contention, several hardware-dependent facilities transparent to the programmer like memory caches become a bottleneck for the throughput of atomic operations and the overall performance of the system. In practice, the introduction of some sort of contention management is indispensable but only allows to absorb the performance impact to some degree.

The real worth of lock-freedom therefore rather lies in its progress guarantees and its implicit solution for all problems usually associated with blocking synchronisation techniques. This includes issues like deadlocks, priority inversion, convoying, and the disastrous consequences of failing to release locks, which all completely vanish by design when using non-blocking algorithms instead.

8.2 Future Work

Our evaluation has identified several aspects where our runtime system could be improved in order to increase its performance. This basically includes all data structures that are prone to high memory contention like the ready queues of the scheduler or the free lists of the heap manager. Although contention management techniques like constant backoff and aligning data to the size of the cache line helped to reduce hardware inefficiencies, the results as well as the comparison with long standing systems indicate that memory contention should be avoided whenever possible.

A simple solution is duplicating all global data structures using processor-local storage such that each processor can naturally access its dedicated portion of the workload without any contention. Examples of this technique include the Linux kernel which uses ready queues dedicated to each processor, or the Hoard memory allocator which allocates memory from superblocks associated with individual processors [BMBW00]. The challenge of this approach lies within the lock-free synchronisation of the processors amongst each other in order to guarantee an evenly distributed workload. Regarding an improved version of the heap manager applying this technique, it is important to ensure that memory always stays available for further allocation if its freed by a different processor than the one that allocated it. A solution to this problem might also improve the currently suboptimal implementation of the coalescing of adjacent free memory blocks.

Another approach to minimise the overall contention programmatically is to extend the task selection model of the scheduler which is currently practically sufficient but rather simplistic in comparison to the feature set provided by schedulers of other operating systems. The introduction of processor affinities for example would allow programmers to explicitly assign tasks with a high potential for memory contention to a small set of dedicated processors. Regarding the targeted simplicity of the runtime system however, it is questionable whether these two approaches discussed above are worth the effort since they presumably would require to replace the comparatively simple ready queues by much more sophisticated lock-free data structures.

Another area needing improvement are the implicit cooperative task switches generated by the non-optimising compiler. Although our measurements have shown that the overhead of the instrumented machine code is in general on par with alternatives based on preemption, we are convinced that an optimising compiler could completely elide the checks in several situations and reduce the overhead even further. The same applies also to other implicitly generated runtime checks like the stack overflow check presented in Section 5.2.

Since there is a relatively strong bond between the runtime system and the compiler, other system components could profit from an optimising compiler as well. For example, an analysing compiler could detect that a reference to an object allocated in one particular procedure never leaves the scope of that procedure. In this case, the allocated object does not need to be registered in the watched list of the garbage collector as it can be deallocated at the exit of the procedure. In fact, the corresponding local variable holding the reference could be treated as untraced and does not require to change the local reference counter of the object.

Conceptually, it would be worthwhile investigating whether our approach of combining cooperative multitasking with lock-free programming also improves the implementation of wait-free algorithms. By solving the problem of potential starvation, wait-freedom would finally allow us to supplement all operations of the runtime system with real-time guarantees.

On a more personal note, it would be interesting to evaluate the comparably high portability of the runtime system even more by bringing it to a plethora of different platforms and embedded systems. Several popular hardware architectures like MIPS and PowerPC based systems come to mind, but also architectures based on message passing like XCore XS1 or even microcontrollers like MicroBlaze which are available exclusively in field-programmable gate arrays.

Appendix

A Digital Material

This dissertation includes a compact disk medium which provides the following digital material:

- The binary code of the runtime system and the compiler targeting AMD64 machines including detailed instructions for installation.
- The complete source code of the runtime system and the compiler including an interactive demo program which offers the following command interface to showcase our runtime system:

  Demo.Line: Moves a random character across the screen.

  Demo.Lines *count*: Executes the above command several times at once by creating *count* threads as a stress test for the scheduler presented in Chapter 4.

  Demo.Allocate *count size*: Allocates and ignores *count* memory blocks containing *size* bytes as a stress test for the heap manager presented in Section 5.1. The resulting memory is deallocated automatically by the garbage collector which is executed at every full minute, if the free memory is exhausted, or by invoking the command **GarbageCollector.Collect**.

  Demo.Insert *count*: Adds *count* elements to a linked list in order to stress the garbage collector while marking the object graph.

  Demo.Clear: Invalidates the reference to the first element of the global list causing all elements to be collected at the next cycle.

  Demo.Recurse: Expands the stack several times as presented in Section 5.2 by performing an infinite recursion causing a trap.

- The source code of all micro-benchmarks evaluated in Chapter 7.

B Module Reference

This appendix describes the interface of all modules shown in Table 6.2 on page 139 and lists their contents in alphabetical order. All information given hereafter was automatically extracted and generated from the source files and is mainly intended for programmers maintaining the system.

B.1 Activities Module

The `Activities` module provides the runtime support for activities associated with active objects. It implements a basic task scheduler that distributes the work of concurrent activities to logical processors. In addition, it also provides a framework for implementing synchronisation primitives.

Interface Summary

Compiler Call	`Activities.Create`
	`Activities.ExpandStack`
	`Activities.Switch`
	`Activities.Wait`
Constant	`Activities.DefaultPriority`
	`Activities.HighPriority`
	`Activities.IdlePriority`
	`Activities.RealtimePriority`
Garbage Collection	`Activities.AssignmentsInProgress`
	`Activities.IsLocalVariable`
Interrupt Handling	`Activities.CallVirtual`
	`Activities.CreateVirtualProcessor`

	Activities.VirtualProcessor
	Activities.awaiting
Procedure	Activities.Call
	Activities.GetCurrentProcessorIndex
	Activities.SetCurrentPriority
	Activities.TerminateCurrentActivity
Runtime Call	Activities.Execute
	Activities.Idle
	Activities.Terminate
Scheduling	Activities.Activity
	Activities.FinalizeSwitch
	Activities.GetCurrentActivity
	Activities.Priority
	Activities.Resume
	Activities.Select
	Activities.SwitchFinalizer
	Activities.SwitchTo

B.1.1 Activities.Activity Object

Represents the handler identifying activities that are currently either running or suspended.

```
TYPE Activity* = OBJECT (VirtualProcessor)
END Activity;
```

B.1.2 Activities.AssignmentsInProgress Procedure

Returns whether any activity is currently executing an assignment statement.

```
PROCEDURE AssignmentsInProgress- (): BOOLEAN;
```

B.1.3 Activities.Call Procedure

Creates a new activity that calls the specified procedure.

```
PROCEDURE Call- (
    procedure: PROCEDURE
);
```

B.1.4 Activities.CallVirtual Procedure

Temporarily exchanges the currently running activity with a virtual processor in order to call the specified procedure in a different context.

```
PROCEDURE CallVirtual- (
    procedure: PROCEDURE (value: ADDRESS);
    value: ADDRESS;
    processor: VirtualProcessor
);
```

B.1.5 Activities.Create Procedure

This procedure is called by the compiler for each NEW statement that creates an active object. It associates an active object with a new activity that begins its execution with the specified body procedure.

```
PROCEDURE Create- (
    body: PROCEDURE;
    priority: Priority;
    object: BaseTypes.Object
);
```

B.1.6 Activities.CreateVirtualProcessor Procedure

Creates an activity that pretends to be executed on a distinct processor.

```
PROCEDURE CreateVirtualProcessor- (): VirtualProcessor;
```

B.1.7 Activities.DefaultPriority Constant

Indicates the default priority of new activities.

```
CONST DefaultPriority* = (* integer value *);
```

B.1.8 Activities.Execute Procedure

Starts the scheduler on the current processor by creating a new activity that calls the specified procedure. This procedure is called by the runtime system once during the initialization of each processor.

```
PROCEDURE Execute- (
    procedure: PROCEDURE
);
```

B.1.9 Activities.ExpandStack Procedure

Expands the stack memory of the current activity to include the specified stack address and returns the corresponding address on the expanded stack.

```
PROCEDURE ExpandStack- (
    address: ADDRESS
): ADDRESS;
```

B.1.10 Activities.FinalizeSwitch Procedure

Finalizes a task switch performed by calling the switch finalizer of the previously suspended activity. This procedure must be called after each invocation of the `Activities.SwitchTo` procedure.

```
PROCEDURE FinalizeSwitch-;
```

B.1.11 Activities.GetCurrentActivity Procedure

Returns the handler of the current activity executing this procedure call.

```
PROCEDURE GetCurrentActivity- (): Activity;
```

B.1.12 Activities.GetCurrentProcessorIndex Procedure

Returns the unique index of the processor executing this procedure call.

```
PROCEDURE GetCurrentProcessorIndex- (): SIZE;
```

B.1.13 Activities.HighPriority Constant

Indicates a higher priority than the default.

```
CONST HighPriority* = (* integer value *);
```

B.1.14 Activities.Idle Procedure

This is the default procedure for initially idle processors starting the scheduler using the `Activities.Execute` procedure.

```
PROCEDURE Idle-;
```

B.1.15 Activities.IdlePriority Constant

Indicates the lowest priority used for idle processors.

```
CONST IdlePriority* = (* integer value *);
```

B.1.16 Activities.IsLocalVariable Procedure

Returns whether the specified address corresponds to a local variable that resides on the stack of the current activity calling this procedure.

```
PROCEDURE IsLocalVariable- (
    address: ADDRESS
): BOOLEAN;
```

B.1.17 Activities.Priority Type

Represents one of four different priorities of an activity.

```
TYPE Priority* = SIZE;
```

B.1.18 Activities.RealtimePriority Constant

Indicates the highest of all priorities.

```
CONST RealtimePriority* = (* integer value *);
```

B.1.19 Activities.Resume Procedure

Resumes the execution of an activity that was suspended by a call to the `Activities.SwitchTo` procedure beforehand.

```
PROCEDURE Resume- (
    activity: Activity
);
```

B.1.20 Activities.Select Procedure

Returns whether there is an activity that is ready to run and has at least the specified priority.

```
PROCEDURE Select- (
    VAR activity: Activity;
    minimum: Priority
): BOOLEAN;
```

B.1.21 Activities.SetCurrentPriority Procedure

Sets the priority of the current activity calling this procedure and returns the previous value.

```
PROCEDURE SetCurrentPriority- (
    priority: Priority
): Priority;
```

B.1.22 Activities.Switch Procedure

Performs a cooperative task switch by suspending the execution of the current activity and resuming the execution of any other activity that is ready to continue. This procedure is called by the compiler whenever it detects that the time quantum of the current activity has expired.

```
PROCEDURE Switch-;
```

B.1.23 Activities.SwitchFinalizer Procedure Type

Represents a procedure that is called after the execution of an activity has been suspended by the `Activities.SwitchTo` procedure.

```
TYPE SwitchFinalizer* = PROCEDURE (
    previous: Activity;
    value: ADDRESS
);
```

B.1.24 Activities.SwitchTo Procedure

Performs a synchronous task switch. The resumed activity continues its execution by first calling the specified finalizer procedure with the given argument. Each invocation of this procedure must be directly followed by a call to the `Activities.FinalizeSwitch` procedure.

```
PROCEDURE SwitchTo- (
    VAR activity: Activity;
    finalizer: SwitchFinalizer;
    argument: ADDRESS
);
```

B.1.25 Activities.Terminate Procedure

Terminates the module and disposes all of its resources.

```
PROCEDURE Terminate-;
```

B.1.26 Activities.TerminateCurrentActivity Procedure

Terminates the execution of the current activity calling this procedure. This procedure is also invoked at the end of the body of an active object.

```
PROCEDURE {NORETURN} TerminateCurrentActivity-;
```

B.1.27 Activities.VirtualProcessor Object

Represents the handler of an activity that resembles a virtual processor.

```
TYPE VirtualProcessor* = OBJECT (Queues.Item)
END VirtualProcessor;
```

B.1.28 Activities.Wait Procedure

This procedure is called by the compiler while executing a `WAIT` statement. It awaits the termination of all activities associated with an active object.

```
PROCEDURE Wait- (
    object: BaseTypes.Object
);
```

B.1.29 Activities.awaiting Variable

Stores an atomic counter indicating the number of activities that are awaiting interrupts to occur. The scheduler stops its execution if all processors are idle, unless there are activities waiting for interrupts.

```
VAR awaiting*: Counters.AlignedCounter;
```

B.2 ExclusiveBlocks Module

The `ExclusiveBlocks` module implements object monitors and provides runtime support for block statements marked as exclusive.

Interface Summary

Compiler Call	`ExclusiveBlocks.Await`
	`ExclusiveBlocks.Enter`
	`ExclusiveBlocks.Exit`
	`ExclusiveBlocks.FinalizeAwait`

B.2.1 ExclusiveBlocks.Await Procedure

Temporarily releases an acquired monitor and waits for other activities to modify the object by entering and exiting an exclusive region. The compiler calls this procedure whenever the repeated evaluation of an `AWAIT` statement yields an unsatisfied condition.

```
PROCEDURE Await- (
    object: BaseTypes.Object
);
```

B.2.2 ExclusiveBlocks.Enter Procedure

Enters an exclusive region by acquiring the monitor of the corresponding object. If the monitor is currently acquired by another activity, this procedure waits until it gets exclusive access to it. The compiler calls this procedure at the beginning of each block statement marked as exclusive. The specified nesting level keeps track of how many times the same activity has acquired the monitor.

```
PROCEDURE Enter- (
    object: BaseTypes.Object;
    nestingLevel: SIZE
);
```

B.2.3 ExclusiveBlocks.Exit Procedure

Exits an exclusive region by releasing the monitor of the corresponding object if its nesting level reaches zero. The compiler calls this procedure at the end of each block statement marked as exclusive or during the execution of a `RETURN` statement inside such blocks. A monitor may not be released if it was not acquired by the same activity beforehand.

```
PROCEDURE Exit- (
    object: BaseTypes.Object;
    nestingLevel: SIZE
);
```

B.2.4 ExclusiveBlocks.FinalizeAwait Procedure

Guarantees that all other activities awaiting a condition will evaluate it again when the current activity leaves the exclusive region. The compiler calls this procedure after the evaluation of an AWAIT statement has yielded a satisfied condition.

```
PROCEDURE FinalizeAwait- (
    object: BaseTypes.Object
);
```

B.3 GarbageCollector Module

The GarbageCollector module provides an automatic memory management for pointers allocated using the NEW statement. It implements a concurrent and interruptible mark and sweep garbage collection.

Interface Summary

Compiler Call	GarbageCollector.Assign
	GarbageCollector.AssignArray
	GarbageCollector.CompareAndSwap
	GarbageCollector.Mark
	GarbageCollector.MarkArray
	GarbageCollector.Reset
	GarbageCollector.ResetArray
	GarbageCollector.Watch
Procedure	GarbageCollector.Collect
Runtime Call	GarbageCollector.Initialize
	GarbageCollector.Terminate

B.3.1 GarbageCollector.Assign Procedure

Performs the assignment of a pointer variable. This procedure is called by the compiler when assigning variables of pointer type.

```
PROCEDURE Assign- (
    VAR pointer: BaseTypes.Pointer;
    value: BaseTypes.Pointer
);
```

B.3.2 GarbageCollector.AssignArray Procedure

Performs the assignment of an array containing pointers. This procedure is called by the compiler when assigning arrays of pointer type.

```
PROCEDURE AssignArray- (
    VAR target: ARRAY OF BaseTypes.Pointer;
    CONST source: ARRAY OF BaseTypes.Pointer
);
```

B.3.3 GarbageCollector.Collect Procedure

Performs a complete garbage collection cycle by marking the object graph and disposing all unreachable objects. Garbage can be collected concurrently if necessary.

```
PROCEDURE Collect*;
```

B.3.4 GarbageCollector.CompareAndSwap Procedure

Executes an atomic compare-and-swap operation on a pointer variable. This procedure is called by the compiler when executing `CAS` expressions.

```
PROCEDURE CompareAndSwap- (
    VAR pointer: BaseTypes.Pointer;
    previousValue: BaseTypes.Pointer;
    value: BaseTypes.Pointer
): BaseTypes.Pointer;
```

B.3.5 GarbageCollector.Initialize Procedure

Initializes the module and its resources.

```
PROCEDURE Initialize-;
```

B.3.6 GarbageCollector.Mark Procedure

Marks the specified pointer as reachable. This procedure is called by the compiler while tracing outgoing pointers of marked objects.

```
PROCEDURE Mark- (
    pointer: BaseTypes.Pointer
);
```

B.3.7 GarbageCollector.MarkArray Procedure

Marks an array of pointers. This procedure is called by the compiler while tracing arrays of pointer type.

```
PROCEDURE MarkArray- (
    CONST pointers: ARRAY OF BaseTypes.Pointer
);
```

B.3.8 GarbageCollector.Reset Procedure

Resets a pointer variable. This procedure is called by the compiler when assigning `NIL` to variables of pointer type.

```
PROCEDURE Reset- (
    VAR pointer: BaseTypes.Pointer
);
```

B.3.9 GarbageCollector.ResetArray Procedure

Resets an array of pointers. This procedure is called by the compiler for resetting all elements of array variables containing pointers.

```
PROCEDURE ResetArray- (
    VAR pointers: ARRAY OF BaseTypes.Pointer
);
```

B.3.10 GarbageCollector.Terminate Procedure

Terminates the module and disposes its resources and all remaining pointers that have been registered using the `GarbageCollector.Watch` procedure.

```
PROCEDURE Terminate-;
```

B.3.11 GarbageCollector.Watch Procedure

Registers the specified pointer for automatic memory management. This procedure is called by the compiler when a pointer or object type is allocated using the `NEW` statement.

```
PROCEDURE Watch- (
    pointer: BaseTypes.Pointer
);
```

B.4 Heaps Module

The `Heaps` module provides a lock-free data structure called *Heap* that handles memory management. A heap manages the allocation and deallocation of blocks of various sizes within a contiguous memory region.

Interface Summary

Procedure	`Heaps.Allocate`
	`Heaps.Deallocate`
	`Heaps.GetSize`
	`Heaps.Initialize`
	`Heaps.IsValid`
Record	`Heaps.Heap`

B.4.1 Heaps.Allocate Procedure

Allocates a block of memory with the requested size from the specified heap. The return value is the first address of the allocated memory, or `NIL` if the heap as no more free memory.

```
PROCEDURE Allocate- (
    size: SIZE;
    VAR heap: Heap
): ADDRESS;
```

B.4.2 Heaps.Deallocate Procedure

Deallocates a memory block that was previously allocated using a call to the `Heaps.Allocate` procedure.

```
PROCEDURE Deallocate- (
    address: ADDRESS;
    VAR heap: Heap
);
```

B.4.3 Heaps.GetSize Procedure

Returns the size of an allocated block of memory.

```
PROCEDURE GetSize- (
    address: ADDRESS;
    CONST heap: Heap
): SIZE;
```

B.4.4 Heaps.Heap Record

Represents a heap which manages a contiguous memory region. Heaps have to be initialised using the `Heaps.Initialize` procedure before they are available for memory allocations.

```
TYPE Heap* = RECORD END;
```

B.4.5 Heaps.Initialize Procedure

Initializes a heap that manages the memory region encompassed by the specified address range. The memory area must be owned by the caller and may not overlap with other heaps. This procedure must be called once before memory can be allocated from the corresponding heap.

```
PROCEDURE Initialize- (
    VAR heap: Heap;
    begin: ADDRESS;
    end: ADDRESS
);
```

B.4.6 Heaps.IsValid Procedure

Checks whether an address is a valid heap address.

```
PROCEDURE IsValid- (
    address: ADDRESS;
    CONST heap: Heap
): BOOLEAN;
```

B.5 Interrupts Module

The `Interrupts` module provides a hardware-independent synchronisation primitive for awaiting the occurrence of interrupts. Activities waiting for an interrupt are suspended and resumed as soon as the interrupt occurred.

Interface Summary

Procedure	`Interrupts.Await`
	`Interrupts.Cancel`
	`Interrupts.Install`
Record	`Interrupts.Interrupt`
Runtime Call	`Interrupts.Terminate`

B.5.1 Interrupts.Await Procedure

Waits for the specified interrupt to occur. This procedure returns as soon as the interrupt has to be handled, or if the wait was cancelled by a call to the `Interrupts.Cancel` procedure.

```
PROCEDURE Await- (
    VAR interrupt: Interrupt
);
```

B.5.2 Interrupts.Cancel Procedure

Resume all activities that are waiting for the specified interrupt.

```
PROCEDURE Cancel- (
    VAR interrupt: Interrupt
);
```

B.5.3 Interrupts.Install Procedure

Installs an interrupt to wait for. The actual meaning of the specified interrupt number identifying the interrupt depends on the hardware.

```
PROCEDURE Install- (
    VAR interrupt: Interrupt;
    index: SIZE
);
```

B.5.4 Interrupts.Interrupt Record

Represents an interrupt.

```
TYPE Interrupt* = RECORD END;
```

B.5.5 Interrupts.Terminate Procedure

Terminates the module and disposes all resources associated with interrupts.

```
PROCEDURE Terminate-;
```

B.6 Processors Module

The `Processors` module represents all logical processors of the system.

Interface Summary

Constant	`Processors.Maximum`
Procedure	`Processors.GetCurrentIndex`
Runtime Call	`Processors.Initialize`
	`Processors.Terminate`
Scheduling	`Processors.ResumeAnyProcessor`
	`Processors.StartAll`
	`Processors.SuspendCurrentProcessor`
Variable	`Processors.count`

B.6.1 Processors.GetCurrentIndex Procedure

Returns the unique index of the processor executing this procedure call.

```
PROCEDURE GetCurrentIndex- (): SIZE;
```

B.6.2 Processors.Initialize Procedure

Initializes the module by enumerating all available processors.

```
PROCEDURE Initialize-;
```

B.6.3 Processors.Maximum Constant

Indicates the maximal number of logical processors that are supported by the system.

```
CONST Maximum* = (* integer value *);
```

B.6.4 Processors.ResumeAnyProcessor Procedure

Resumes the execution of a single suspended processor. This procedure may not be called if there is only one processor in the system.

```
PROCEDURE ResumeAnyProcessor-;
```

B.6.5 Processors.StartAll Procedure

Starts the execution of all available processors. This procedure may not be called if there is only one processor in the system.

```
PROCEDURE StartAll-;
```

B.6.6 Processors.SuspendCurrentProcessor Procedure

Suspends the execution of the current processor. A suspended processor must be resumed by a call to the `Processors.ResumeAnyProcessor` procedure. This procedure may not be called if there is only one processor in the system.

```
PROCEDURE SuspendCurrentProcessor-;
```

B.6.7 Processors.Terminate Procedure

Terminates the module and waits for all other processors to stop their execution.

```
PROCEDURE Terminate-;
```

B.6.8 Processors.count Variable

Holds the actual number of processors in the system.

```
VAR count-: SIZE;
```

B.7 Queues Module

The `Queues` module provides a lock-free data structure called *Queue*. A queue is a sequential container which allows to append elements at the back of it and to remove the elements at its front.

Interface Summary

Object	`Queues.Item`
Procedure	`Queues.Dequeue`
	`Queues.Dispose`
	`Queues.Enqueue`
Record	`Queues.AlignedQueue`
	`Queues.Queue`
Runtime Call	`Queues.Terminate`

B.7.1 Queues.AlignedQueue Record

Represents a first-in first-out data structure which is aligned for optimal cache behavior.

```
TYPE AlignedQueue* = RECORD (Queue) END;
```

B.7.2 Queues.Dequeue Procedure

Removes the first element at the front of a queue and returns it in a variable parameter. If there are no elements in the queue, the procedure returns `FALSE` and sets the variable parameter to `NIL`.

```
PROCEDURE Dequeue- (
    VAR item: Item;
    VAR queue: Queue
): BOOLEAN;
```

B.7.3 Queues.Dispose Procedure

Disposes the elements of a queue.

```
PROCEDURE Dispose- (
    VAR queue: Queue
);
```

B.7.4 Queues.Enqueue Procedure

Appends an element at the back of a queue.

```
PROCEDURE Enqueue- (
    item: Item;
    VAR queue: Queue
);
```

B.7.5 Queues.Item Object

Represents an abstract element of a queue.

```
TYPE Item* = OBJECT
END Item;
```

Procedure `Item.Finalize`

Queues.Item.Finalize Procedure

Finalizes the element by disposing any resources associated with it.

```
PROCEDURE Finalize-;
```

B.7.6 Queues.Queue Record

Represents a first-in first-out data structure.

```
TYPE Queue* = RECORD END;
```

B.7.7 Queues.Terminate Procedure

Terminates the module and disposes all of its resources.

```
PROCEDURE Terminate-;
```

C Enhancements to the Active Oberon Programming Language

Any implementation of an operating system that targets multiprocessing environments requires some sort of specialised hardware support. However, one of the most important objectives of this thesis was to maximise the portability of its source code across different hardware architectures. The use of a high-level programming language allows to access the necessary hardware features in a portable way. By this means, the corresponding hardware abstraction layer can be provided by the implementation of the programming language and related software development tools.

The current implementation of the lock-free runtime system presented in this thesis was written using the Active Oberon programming language [Rea04]. Although this language provides a simple yet powerful base for the necessary hardware abstraction, some important features have still been missing. This appendix summarises all changes and additions to the Active Oberon programming language that were necessary in order to achieve full machine independence. It therefore assumes some basic familiarity with Active Oberon and its development tools. Most of the modifications address portability issues that hindered the same source code from being compilable for different targets. Other enhancements concern inconvenient shortcomings or minor oversights in the initial design of the programming language.

Each issue listed in the following sections is accompanied by a short rationale for the modification and provides informative examples for the enhanced syntax or semantics where possible. However, the information in this appendix is not intended to replace the original language specification. The issues raised hereafter are rather suggestions to take into consideration in a potential future rendition of the Active Oberon report.

C.1 Atomic Compare-And-Swap

All data structures and algorithms employed in this thesis are based on the atomic compare-and-swap operation. It is the only atomic operation that has to be provided by the hardware architecture.

Change: Added a new built-in `CAS` procedure for atomically comparing and exchanging the values of shared variables.

Syntax: `PROCEDURE CAS (VAR variable: T; old, new: T): T;`

Semantics: This procedure compares the value of the variable named in the first argument with the value of the second argument. If the two values of basic or reference type match, the variable is overwritten with the value of the third argument. The result is equal to the original value of the variable. The whole operation is executed atomically and never interrupted by any other activity. If the second and third argument are the same, the whole operation effectively equals to an atomic read of a shared variable.

Example: Other atomic operations like test-and-set can be implemented on top of the `CAS` procedure:

```
PROCEDURE TAS* (VAR value: BOOLEAN): BOOLEAN;
BEGIN RETURN CAS (value, FALSE, TRUE);
END TAS;
```

C.2 Memory Model

The original Active Oberon report does not specify how concurrent activities interact when operating on shared variables without proper synchronisation. A memory model defines how shared memory behaves in this case and is crucial for lock-free programming. Having no guarantees for non-blocking algorithms, programmers often made implicit assumptions about the implementation chosen by their particular compiler.

Change: Added the definition of a simple memory model which provides basic guarantees required by portable non-blocking programs.

Semantics: The memory model requires that concurrent access to shared data must be either protected by exclusive blocks or performed by an atomic compare-and-swap operation. The detailed rules are specified in Section 2.2.2.

C.3 Variable Initialisation

The original language specification assures a default value of `NIL` for all local variables of reference type. This hinders pointers from referencing arbitrary memory and thus allows effective `NIL` pointer checks. However, all other variables have no default value and store an arbitrary value if not initialised properly.

Change: Variables in modules, procedures, records, and objects can be automatically initialised with a constant expression.

Syntax: `IdentDef = ident ["*"|"-"] [":=" ConstExpr].`

Semantics: The optional constant initialiser of a variable works as if there was an assignment to that variable at the beginning of the body of the corresponding scope. The value has therefore to be compatible with the type of the variable. Fields of records can be initialised the same way. Variables of objects are initialised before the execution of their initialiser. The original rule for variables of reference type still holds because they have an implicit `:= NIL` initialiser.

Example: Initialisers for variables of record types often spare an explicit call to a corresponding procedure that ensures a proper initialisation and make the code shorter and safer:

```
TYPE Counter* = RECORD value := 0: INTEGER END;
```

C.4 Accessing Externally Defined Symbols

Sometimes it is necessary to access data or procedures that are provided by a different software development tools. This includes external application programming interfaces or low-level functionality written in assembly code or other programming languages.

Change: Global variables and procedures may be declared as `EXTERN`.

Syntax: `IdentDef = ident ["*"|"-"] ["EXTERN" string].`

An external procedure does not have a procedure body or a declaration sequence. Its syntactical definition is completed after the semicolon delimiting the procedure signature.

Semantics: The declaration of an external variable or procedure does not generate any actual data or code. Accessing the value of an external variable or calling an external procedure actually refers to the symbol named in the string literal following the `EXTERN` keyword.

Example: Low-level code often needs access to data structures which are initialised by assembly programs. An interrupt vector table provided by a low-level bootloader serves well as an illustrative example:

```
MODULE CPU;
TYPE Vector = ARRAY 32 OF PROCEDURE;
VAR interrupts* EXTERN "vector_table": Vector;
END CPU.
```

C.5 Portable Integer Types

Active Oberon provides four integer types: `SHORTINT`, `INTEGER`, `LONGINT`, and `HUGEINT`. All of them are signed and represent integral values with preassigned bit widths of 8, 16, 32, and 64 respectively. However, there is no notion of an integral type that has the default integer bit width of the target hardware architecture.

In the course of time, the lack of such a distinguished type led to an overuse of the `LONGINT` type. Depending on context, this may be acceptable for code that runs on 32-bit machines. But regarding machines with a default word size of 16 bits, `LONGINT` type could be excessive. On architectures with 64 bits words on the other hand, it may be misrepresentative.

Change: Complemented the integral types with two new integer types called `WORD` and `LONGWORD` which have a machine-dependent bit width.

Syntax: `Type = ... | WORD | LONGWORD.`

Semantics: The type `WORD` is merely a synonym for an existing integral type that fits best to the default word size of the target machine. On 8-bit and 16-bit machines, the type typically equals the `INTEGER` type. On 32-bit and 64-bit machines, the type is often equal to `LONGINT`. The type `LONGWORD` is a synonym for an existing integral type that fits best to the default address size of the target machine.

Example: The new types are very useful when accessing functionality that is provided by a different software environment like for example assembly code or application programming interfaces:

```
PROCEDURE {NORETURN} Exit EXTERN "exit" (status: WORD);
```

For the very same reason as above, the type `LONGINT` was also abused for representing addresses and memory ranges in low-level code. This is obviously only valid for hardware architectures with a 32-bit address space. Without a proper type for representing memory addresses with differing bit widths, the corresponding source code cannot be ported easily to other architectures. Additionally, the signed nature of the integral types render comparison operations on memory addresses highly questionable. In cases where memory ranges are represented in a high-level fashion using arrays, the same issues also arise concerning the type of the array length and the indices used to access individual elements.

Change: Added the unsigned types ADDRESS and SIZE for representing addresses and array lengths with the proper bit width.

Syntax: `Type = ... | ADDRESS | SIZE.`

Semantics: Both types are distinct integral types that are not compatible with any other basic type. Signed integral types are compatible with both types if they have the same or lower bit width.

The SIZE type allows all integral operations and is necessary for specifying array lengths and indices. The unary SIZE OF operator returns the size of any basic or reference type and is useful to determine the word or address size of the machine.

The type ADDRESS only allows comparison operations as well as addition with a SIZE type yielding another address, while the difference of two addresses yields a SIZE type. All reference types are implicitly convertible to the ADDRESS type and yield the address of the referenced object. The address of a variable or procedure can be determined using the unary ADDRESS OF operator.

Example: The new SIZE type allows to represent all kinds of lengths:

```
PROCEDURE GetLength (CONST string: ARRAY OF CHAR): SIZE;
VAR length := 0: SIZE;
BEGIN
  WHILE string[length] # 0X DO INC (length) END;
  RETURN length;
END GetLength;
```

C.6 Unified Debugging Facilities

The only built-in debugging facilities of Active Oberon are the ASSERT and HALT statements. They allow to check whether preconditions of procedures or other invariants are satisfied at runtime. However, in order to print values of expressions for debugging purposes, programmers often use various and inconsistent output methods which have a typically cumbersome interface.

Change: Added a new built-in TRACE procedure for convenient debugging.

Syntax: PROCEDURE TRACE (expr1: T1 {; exprN: TN});

Semantics: The TRACE procedure prints arbitrary expressions, the value thereof, as well as the corresponding location in the source code to the standard output. If an expression is of basic or reference type, the TRACE procedure prints a textual representation of the corresponding value. For structured types, the printed value is equal to the address of the first element. Concurrent calls to the TRACE procedure are sequentialised yielding a consistent view on the data.

Example: The TRACE procedure is easy to use, unifies the format of all debugging outputs, and is highly recognisable in code reviews. A call to the Add procedure defined below generates the following diagnostic message:

```
module ‘Test’ procedure ‘Add’ line 6:
‘x’ = 4, ‘y’ = 8, ‘result’ = 12, ‘result = x + y’ = TRUE
```

```
MODULE Test;
  PROCEDURE Add* (x, y: WORD): WORD;
  VAR result: WORD;
  BEGIN
    result := x + y;
    TRACE (x, y, result, result = x + y);
    RETURN result;
  END Add;
END Test.
```

C.7 Unsafe Pointers

Typically, low-level Active Oberon code makes heavy use of the built-in procedures SYSTEM.PUT, SYSTEM.GET, and variations thereof. The goal is most often to manipulate data structures which cannot be accessed directly using standard language features. Examples include meta data generated by the compiler or data structures defined by the hardware.

Change: Pointer types can be annotated as `UNSAFE` in order to access arbitrary memory using default language features.

Syntax: `Type = ... | "POINTER" ["{" "UNSAFE" "}"] TO Type.`

Semantics: A variable of type `ADDRESS` can be assigned to a variable of pointer type annotated as `UNSAFE`. The referenced memory can therefore be accessed as if it was mapped as a record or an array. Unsafe pointers are not compatible with default pointer types.

Example: The stack frames of a call stack can be portably accessed by an unsafe pointer to record:

```
PROCEDURE TraceCallStack*;
TYPE StackFrame = RECORD previous, caller: ADDRESS END;
VAR frame: POINTER {UNSAFE} TO StackFrame;
BEGIN
  frame := ADDRESS OF frame + SIZE OF ADDRESS;
  REPEAT
    TRACE (frame.caller);
    frame := frame.previous;
  UNTIL frame = NIL;
END TraceCallStack;
```

C.8 Manual Memory Deallocation

Active Oberon as specified in its original report relies on a garbage collector that manages all resources allocated from free storage. However, garbage collection may be unsuitable for software like real-time applications because of its non-deterministic delays and execution overhead. In order to provide an execution environment for this kind of software, our runtime system was designed to be able to optionally omit the automatic memory management completely. It is therefore necessary to extend the programming language to allow programmers to deallocate resources by hand. The runtime system itself always manages all of its resources manually and does hence not rely on garbage collection.

Change: Added a new built-in DISPOSE procedure that deallocates an object from free storage.

Syntax: `PROCEDURE DISPOSE (VAR variable: PTR);`

Semantics: The value passed to this procedure must be an object or a pointer to an array or record that was previously allocated using the NEW procedure. The variable is reset to the value NIL after the memory is released to free storage. The reference type has to be marked explicitly with the DISPOSABLE modifier in order to enable manual memory deallocation on instances of this type.

Example: Garbage collection spares the developer from manual memory deallocation but can still suffer from memory leaks if unused objects stay reachable long enough. If applied properly, the DISPOSE procedure does not leak memory and releases it immediately:

```
PROCEDURE Free (VAR tree: POINTER {DISPOSABLE} TO Node);
BEGIN
  IF tree.left # NIL THEN Free (tree.left) END;
  IF tree.right # NIL THEN Free (tree.right) END;
  DISPOSE (tree);
END Free;
```

C.9 Object Finalizers

The Active Oberon report defines object initialisers which are procedures that are automatically called whenever an object is created. Regarding the symmetry of object allocation and deallocation, the language lacks an equivalent mechanism for automatically calling a procedure whenever an object is destroyed.

Change: Methods tagged with a tilde are object finalisers.

Syntax: `ProcHead = ["&" | "~"] IdentDef [FormalPars].`

Semantics: Object finalisers are automatically called once when an object instance is deallocated using the built-in DISPOSE procedure. They follow the same set of rules defined for object initialisers except that they may not have any parameters in their signature.

Example: Object finalisers allow to deallocate resources managed by active objects whose lifetime is non-deterministic:

```
TYPE Driver* = OBJECT
  VAR buffer: POINTER {DISPOSABLE} TO ARRAY OF CHAR;

  PROCEDURE &Initialize*;
  BEGIN NEW (buffer, 1024);
  END Initialize;

  PROCEDURE ~Finalize*;
  BEGIN DISPOSE (buffer);
  END Finalize;

BEGIN {ACTIVE}
  ...
END Driver;
```

C.10 Proper Synchronisation of Active Objects

Sometimes it is necessary to wait for an active object to complete its active body. This kind of synchronisation is achieved by using the monitor of the active object to await a condition that is only satisfied at the very end of the active body. Although this technique is often sufficient in practice, it does strictly speaking not guarantee that the activity has actually completed its execution. This deficiency becomes apparent when resources like the call stack are released because the active object is erroneously assumed to have completed its active body.

Change: Added a new built-in WAIT procedure for waiting on the termination of active objects.

Syntax: `PROCEDURE WAIT (object: OBJECT);`

Semantics: A call to this procedure awaits the termination of all activities associated with an active object. An activity terminates, if it successfully completes its execution or aborts it because of an exception. This procedure effectively guarantees that an object is passive. It may not be called within the initialiser or body of an active object because the active body can never run to completion.

Example: An active object can be safely deallocated after its termination:

```
PROCEDURE Work;
VAR object: OBJECT {DISPOSABLE} BEGIN {ACTIVE} ... END;
BEGIN
  NEW (object);
  ...
  WAIT (object);
  DISPOSE (object);
END Work;
```

C.11 Improved Handling of Runtime Errors

Recent implementations of Active Oberon added a simplistic form of exception handling in the form of the `FINALLY` statement [Sze05]. It allows to catch and react properly to traps generated by the runtime system. Examples of such failures include unsatisfied assertions, referencing a `NIL` pointer, or using an array index that is out of range. However, all of these traps indicate programming errors and it is highly questionable whether they should be handled at all.

Change: Dropped the support for the `FINALLY` statement.

Semantics: Without loss of generality, any code protected by a `FINALLY` statement can be executed by a special-purpose active object. The original code including the `FINALLY` statement can then be replaced by code that creates that active object and waits for it to terminate.

Example: The following procedure calls a procedure variable and returns whether the call succeeded or resulted in a runtime failure:

```
PROCEDURE CallSafely* (procedure: PROCEDURE): BOOLEAN;
VAR caller: Caller; result: BOOLEAN;
BEGIN
  NEW (caller, procedure); WAIT (caller);
  result := caller.succeeded;
  DISPOSE (caller); RETURN result;
END CallSafely;

TYPE Caller = OBJECT {DISPOSABLE}
  VAR procedure: PROCEDURE;
  VAR succeeded := FALSE: BOOLEAN;

  PROCEDURE &Initialize (procedure: PROCEDURE);
  BEGIN SELF.procedure := procedure;
  END Initialize;

BEGIN {ACTIVE}
  procedure; (* this procedure may fail *)
  succeeded := TRUE;
END Caller;
```

C.12 Uncooperative Blocks

The implementation of the scheduler is based on implicit cooperative task scheduling. Task switches are emitted by the compiler at various places in the compiled code as described in Section 4.2.4. In order to implement the task switches themselves using the very same software development tools, it is necessary to be able to temporarily disable the emission of task switches.

Change: Introduced the `UNCOOPERATIVE` modifier for statement blocks.

Syntax: `StatBlock = "BEGIN" ["{" "UNCOOPERATIVE" "}"] ...`

Semantics: The compiler omits all implicit task switches in a statement block annotated with the `UNCOOPERATIVE` modifier. This has the effect that the executed code does never voluntarily give up the control nor cooperatively pass it to another activity.

Example: All lock-free algorithms profit from uncooperative blocks because they limit the maximal number of activities that concurrently execute the code. This offers several advantages which are described in detail in Section 2.3.1. The following atomic increment operation for instance is based on the algorithm shown in Listing 2.2 on page 26 and cannot be executed by more concurrent activities than there are processors in the system:

```
PROCEDURE Increment* (VAR counter: Counter): INTEGER;
VAR previous, value: INTEGER;
BEGIN {UNCOOPERATIVE}
  REPEAT
    previous := CAS (counter.value, 0, 0);
    value := CAS (counter.value, previous, previous + 1);
  UNTIL value = previous;
  RETURN previous;
END Increment;
```

C.13 Unchecked Blocks

The compiler generates several different implicit runtime checks that prevent programs from corrupting or accessing invalid memory. This most prominently includes null pointer and array bound checks as well as stack overflow checking as described in Section 5.2. In order to be able to implement these runtime checks using the same compiler without causing infinite recursion, it is necessary to temporarily disable their generation similar to uncooperative blocks.

Change: Introduced the `UNCHECKED` modifier for statement blocks.

Syntax: `StatBlock = "BEGIN" ["{" "UNCHECKED "}"] ...`

Semantics: The compiler does not generate any runtime checks inside an unchecked statement block and completely relies on the programmer to provide correct code. Explicit checking for erroneous conditions in an unchecked block remains possible using the ASSERT statement.

Example: Any procedure called while expanding an exceeded stack must be unchecked in order to prevent the compiler from expanding the stack recursively. One important operation thereof is copying the contents of the old stack into the new and enlarged stack memory as described in Section 5.2. The following type represents individual stack frames in the new stack memory which can be used to adapt all pointers into the old stack. The compiler provides extensions of this type for procedures with variable parameters and overwrites the Move method accordingly:

```
TYPE StackFrame* = OBJECT {UNSAFE}

  VAR descriptor-: ADDRESS;
  VAR previous-: ADDRESS;
  VAR caller-: PROCEDURE;

  PROCEDURE Move* (offset: SIZE);
  BEGIN {UNCOOPERATIVE, UNCHECKED}
    IF previous # NIL THEN INC (previous, offset) END;
  END Move;

END StackFrame;
```

C.14 Reentrant Exclusive Regions

The original Active Oberon report states that an activity cannot enter an exclusive region more than once. Experience with existing Active Oberon programs has shown however, that this artificial restriction misled developers into duplicating a lot of protected procedures. The only difference of the duplicated code is the omission of the exclusive block modifier.

Change: Explicitly allow reentrancy into exclusive regions.

Semantics: The same activity may enter an exclusive region more than once. The corresponding protected object is locked until the activity leaves the outermost exclusive block. If the condition of an AWAIT statement is not satisfied, the lock on the protected object is released completely.

Example: Whether or not a supercall to an overridden method is actually valid according to the old semantics, depends on whether the overriding as well as the overridden procedures enter an exclusive region simultaneously. With the relaxed rule, the developer of an overriding method can always enter an exclusive region if necessary regardless of whether a subsequent supercall might do the same. It is therefore not required to know the actual implementation of the overridden procedure any more:

```
TYPE Base = OBJECT

  PROCEDURE Method*;
  BEGIN {EXCLUSIVE}
    ...
  END Method;

END Base;

TYPE Extension = OBJECT (Base)

  PROCEDURE Method*;
  BEGIN {EXCLUSIVE}
    ...
    Method^;
    ...
  END Method;

END Extension;
```

List of Listings

List of Figures

List of Tables

Bibliography

[ACHS14] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are Lock-Free Concurrent Algorithms Practically Wait-Free? In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, 2014.

[Adv13] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, May 2013. Revision 3.23.

[And90] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[ARM05] ARM Limited. *ARM Architecture Reference Manual*, July 2005. Issue I.

[Atm05] Atmel Coorperation. *8-bit AVR Instruction Set*, November 2005. Revision E.

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly Garbage Collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, 1984.

[BDF92] Brian Bershad, Richard P. Draves, and Alessandro Forin. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, 1992.

[Ber02] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, University of Texas at Austin, 2002.

[Bia98] Gabriele Bianchi. Incremental Garbage Collector for XOberon. Diploma thesis, ETH Zurich, 1998.

[Blä07] Luc Bläser. *A Component Language for Pointer-Free Concurrent Programming and its Application to Simulation*. PhD thesis, ETH Zurich, 2007.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'00, 2000.

[Con63] Melvin E. Conway. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[Cre94] Régis B. J. Crelier. *Separate Compilation and Module Extension*. PhD thesis, ETH Zurich, 1994.

[Dah99] Ole-Johan Dahl. A Note on Monitor Versions. In *Proceedings of the Symposium in the Honour of C. A. R. Hoare at his Resignation from the University of Oxford*, 1999.

[DG02] Dave Dice and Alex Garthwaite. Mostly Lock-Free Malloc. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM'02, 2002.

[DHM13] David Dice, Danny Hendler, and Ilya Mirsky. Lightweight Contention Management for Efficient Compare-and-Swap Operations. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, 2013.

[Die92] Reinhard A. Dietrich. Dynamische Stacks für Lightweightprozesse. Diploma thesis, ETH Zurich, 1992.

[DMSJ02] David L. Detlefs, Paul A. Martin, and Guy L. Steele Jr. Lock-Free Reference Counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, PODC'02, 2002.

[ECM06] ECMA. *Standard ECMA-334 C# Language Specification*. ECMA International, 2006.

[Egg01] Bernhard Egger. Development of an Aos Operating System for the DNARD Network Computer. Diploma thesis, ETH Zurich, 2001.

[FH82] Christopher W. Fraser and David R. Hanson. A Machine-Independent Linker. *Software — Practice and Experience*, 12(4):351–366, 1982.

[FN11] Felix Friedrich and Florian Negele. A Compiler-Supported Unification of Static and Dynamic Loading. In *Tagungsband 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, KPS'11, 2011.

[Fog14] Agner Fog. *The Microarchitecture of Intel, AMD and VIA CPUs*. Technical University of Denmark, Feburary 2014.

[Fra04] Keir Fraser. Practical lock-freedom. Technical report, Computer Laboratory, University of Cambridge, 2004.

[Fri11] Felix Friedrich. Flexible Oberon Cross-Compiler (FOX). Implementation report, Institute of Computer Systems, ETH Zurich, 2011.

[GC96] Michael Greenwald and David Cheriton. The Synergy Between Non-Blocking Synchronization and Operating System Structure. In *Second Symposium on Operating Systems Design and Implementation*, OSDI'96, 1996.

[GGH07] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-Free Parallel Garbage Collection by Mark&Sweep. *Science of Computer Programming*, 64(3):341–374, 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[Glo06] Wolfram Gloger. Wolfram Gloger's malloc homepage. See `http://www.malloc.de/`, 2006.

[GP07] Anders Gidenstam and Marina Papatriantafilou. LFTHREADS: A Lock-Free Thread Library. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, 2007.

[GPST09] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009.

[GPT05] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating Memory in a Lock-Free Manner. In *Proceedings of the 13th Annual European Conference on Algorithms*, ESA'05, 2005.

[Gre99] Michael Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[Gut97] Jürg Gutknecht. Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon. In *Modular Programming Languages*, volume 1204 of *Lecture Notes in Computer Science*, pages 207–220. Springer Berlin Heidelberg, 1997.

[HB84] Kai Hwang and Fayé Alayé Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[Her88] Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'88, 1988.

[Her90] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPOPP'90, 1990.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[HG01] Wim H. Hesselink and Jan Friso Groote. Wait-Free Concurrent Memory Management by Create and Read Until Deletion. *Distributed Computing*, 14(1):31–39, 2001.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic Nonblocking Synchronization for Realtime Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, USENIX'01, 2001.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS'03, 2003.

[HLMM02] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-Sized Lock-Free Data Structures. In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing*, PODC'02, 2002.

[HM91] Maurice Herlihy and J. Eliot B. Moss. Lock-Free Garbage Collection for Multiprocessors. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'91, 1991.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, 1993.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Elsevier Science, 2008.

[HW90] Maurice Herlihy and Jeanette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[IBM83] IBM Corporation. *IBM System/370 Extended Architecture Principles of Operation*, 1983. Publication Number SA22-7085-0.

[Int14] Intel Cooperation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, February 2014. Order Number: 253665-049US.

[ISO95] ISO. *ISO/IEC 8652:1995: Information technology — Programming languages — Ada*. International Organization for Standardization, 1995.

[ISO11a] ISO. *ISO/IEC 9899:2011: Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[ISO11b] ISO. *ISO/IEC 14882:2011: Information technology — Programming languages — C++*. International Organization for Standardization, 2011.

[Kno65] Kenneth C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[Knu68] Donald E. Knuth. Dynamic Storage Allocation. In *The Art of Computer Programming*, volume 1, pages 435–455. Addison-Wesley, 1968.

[KP11] Alex Kogan and Erez Petrank. Wait-Free Queues With Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'11, 2011.

[KP12] Alex Kogan and Erez Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, 2012.

[Lov10] Robert Love. *Linux Kernel Development, Third Edition*. Addison-Wesley Professional, 2010.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[MB76] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Network. *Communications of the ACM*, 19(7):395–404, 1976.

[Mel87] John M. Mellor-Crummey. Concurrent Queues: Practical Fetch-and-Φ Algorithms. Technical Report 229, Computer Science Departement, University of Rochester, 1987.

[Mic04a] Maged M. Michael. ABA Prevention Using Single-Word Instructions. Research report, IBM Research Division, 2004.

[Mic04b] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mic04c] Maged M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI'04, 2004.

[Mic10] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*, September 2010. Revision 8.2.

[Mol07] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler [CFS]. See `http://lwn.net/Articles/230501/`, 2007.

[MP91] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. Technical report, Department of Computer Science, Columbia University, 1991.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical report, Computer Science Departement, University of Rochester, 1995.

[MS96] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, 1996.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[Mul02] Pieter J. Muller. *The Active Object System Design and Multiprocessor Implementation*. PhD thesis, ETH Zurich, 2002.

[Neg06] Florian Negele. Porting the Active Oberon System to the AMD64 Architecture. Master's thesis, ETH Zurich, 2006.

[Rea03] Patrik Reali. *Using Oberon's Active Objects for Language Interoperability and Compilation*. PhD thesis, ETH Zurich, 2003.

[Rea04] Patrik Reali. Active Oberon Language Report. Technical report, Institute of Computer Systems, ETH Zurich, 2004.

[RG06] Patrik Reali and Ulrike Glavitsch. Aos/Bluebottle: Symbol and Object File Format. Technical report, Institute of Computer Systems, ETH Zurich, 2006.

[RSI12] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Sixth Edition*. Microsoft Press, 2012.

[Sze05] Michael Szediwy. Aos Thread Finalization & Termination. Diploma thesis, ETH Zurich, 2005.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[Too95] Tool Interface Standard (TIS). *Executable and Linking Format (ELF) Specification*, May 1995. Version 1.2.

[Val94] John D. Valois. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, PDCS'94, 1994.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[Wir88] Niklaus Wirth. The Programming Language Oberon. *Software — Practice and Experience*, 18(7):671–690, 1988.