Self-Stabilizing Operating Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of
"DOCTOR OF PHILOSOPHY"

by

Reuven Yagel

Submitted to the Senate of Ben-Gurion University
of the Negev

September 2007

Beer-Sheva

Self-Stabilizing Operating Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of
"DOCTOR OF PHILOSOPHY"

by

Reuven Yagel

Submitted to the Senate of Ben-Gurion University
of the Negev

Approved by the advisor: ___________

Approved by the Dean of the Kreitman School of Advanced Graduate Studies: ___________

September 2007

Beer-Sheva

This work was carried out under the supervision of Prof. Shlomi Dolev

In the Department of Computer Science

Faculty of Natural Science

Acknowledgments

This work could not have been completed without the aid of many people. First of all my advisor Professor Shlomi Dolev who introduced me to the worlds of self-stabilization, distributed computing and academic research in general. We spent many hours together suggesting and validating ideas, and also having enjoyable moments of friendship. Whenever I thought I reached a dead end, he was there to patiently find the way out. Thanks also to the people behind the research funds achieved by him.

Ben-Gurion University and the Computer-Science department were an excellent place for doing this research. My thanks to all faculty, administrative and technical staff. I would like to thank Meni Adler for continuous friendship and remote aid, and the DCCN group members: Ronen Kat, Nir Tzachar, Yinon Haviv, Olga Brukman, Limor Lahiani, Hen Fitoussi, Marina Kopeetsky, Noam Singer, Elad Schiller, Ori Gersten and Rami Puzis for sharing ideas and helpful feedback. Dan Epelman, Edna Oxman and especially my father served as excellent English editors. My PhD proposal committee contributed also to shaping this research, and also the anonymous thesis reviewers. I would like to thank Sivan Toledo, Amnon Meisels and notably Michael Elhadad for guidance and inspiration.

While doing this research I also worked at Rafael. Many thanks to Aharon Leshem and Arie Rozengart, my managers who supported starting this work. Many other, were helpful by exchanging ideas, inspiration, or even just by providing a friendly and supporting environment. Rachel Mor and Motti Hooper were also very helpful with administrative issues.

This is the place to thank my parents, my parents-in-law and all brothers, sisters, in-laws and the broader family for continuous support. Many friends also showed their big hearts during this period. In particular, Yuval Feldman, Menachem Geva and the Mor-Yosefs from Qazrin who helped me cross the river on a daily basis.

Thank you all!

Dedication

This work is dedicated to my beloved wife Yifat and children who supported and followed this long journey and in loving memory of Achiya who couldn't make it to the end.

Contents

List of Figures vi

List of Tables vii

Abstract ix

1 Introduction 1

1.1 Motivation 1
1.2 Thesis and Contribution 4
1.3 Work Organization. 7

2 Related Work 9

2.1 Historical Brief Review 9
2.2 Robust System Efforts 10
2.3 Hardware Support 10
2.4 Minimalistic Approaches 11
2.5 Fault Tolerance Mechanisms 12
2.6 Fault Avoidance 12
2.7 Other Related Research 13
2.8 Summary 13

3 System Models and Assumptions 15

3.1 The System Model 15
3.2 Additional Necessary and Sufficient Hardware Support 17
3.3 The Error Model 19

4 Blackbox Techniques 21

4.1 Periodical Reinstall and Restart 21
4.2 Reinstall Executable and Monitor State 25
4.3 Concluding Remarks 25

5 Process Scheduling 27

5.1 Introduction, Settings and Requirements 27
5.2 A Primitive Scheduler 28

5.3 A Self-Stabilizing Scheduler 29
5.4 Concluding Remarks 36

6 Memory Management 37
6.1 Introduction 37
6.2 System Settings, Assumptions and Requirements 39
6.3 Total Swapping — One Process at a Time 40
6.4 Fixed Partition — Multiple Residing Processes 43
6.5 Dynamic Allocation 47
6.6 Concluding Remarks 53
6.7 Appendix. Implementations 54

7 I/O Device Drivers 63
7.1 Introduction 63
7.2 A Non-Self-Stabilizing Driver Specification 66
7.3 System Model and Requirements 68
7.4 A Self-Stabilizing Driver 70
7.5 Concluding Remarks 74
7.6 Appendix. Hard-Disk Driver Implementation 75

8 Stabilizing Hosts 87
8.1 Introduction 87
8.2 Settings and the Requirements 91
8.3 Concepts for Fighting the Byzantines 91
8.4 Host Implementation Example 93
8.5 Concluding Remarks 98

9 Concluding Remarks 99

Bibliography 101

List of Figures

4.1 Operating System Watchdog/Reinstall Procedure 22

5.1 System Transitions 31
5.2 Refresh Fixed Addresses and Store `ax,bx,ds` Values Towards Sequel Use 32
5.3 Save Process State 33
5.4 Increment Process 33
5.5 Load Process State 34

6.1 A Common Memory Hierarchy 39
6.2 Total Swapping Algorithm 41
6.3 Fixed Partition Consistency Check 43
6.4 Fixed Partition Algorithm 45
6.5 Dynamic Allocation Services 49
6.6 Dynamic Allocation Validation 50
6.7 Dynamic Allocation Service Procedures 51
6.8 Total Swapping Implementation 54
6.9 Fixed and Dynamic Validator Implementations 60
6.10 Dynamic Memory Implementation 61

7.1 ATA Host State Transitions. 66
7.2 ATA Device State Transitions. 66
7.3 Union State Transitions (h=host, d=device). 68
7.4 A Leasing Host (OS Driver). 71
7.5 HD Driver Impl. 1-2 76
7.6 HD Driver Impl. 3-4 77
7.7 HD Driver Impl. 5-6 80
7.8 HD Driver Impl. 7-8 81
7.9 HD Driver Impl. 9-10 82
7.10 HD Driver Impl. 11-12 83
7.11 HD Driver Impl. 13-14 84
7.12 HD Driver Impl. 15-16 85
7.13 HD Driver Impl. 17 - Failure Handler and Lease Procedures. . . 86

8.1 Byzantine Code. 94
8.2 Memory Access Enforcer. 95
8.3 Shared Resource Access. 95
8.4 An Allowed Pattern. 96
8.5 Out of Segment Access Detector 96
8.6 Update Trust and Reputation – Increase and Decay 97

List of Tables

7.1 ATA Registers. 70
7.2 Consistency Check Rules. 73

8.1 Byzantine Threats. 89

Abstract

Interest in robust system design has increased in recent years. One reason is the increase of vulnerability to soft-errors. A novel way to deal with such threats is by designing a system to be self-stabilizing. The self-stabilization property captures the desire to recover automatically from any (unexpected) state. A self-stabilizing system converges to a legal execution once faults stop occurring.

This work presents new concepts and directions for building a self stabilizing operating system kernel. A self-stabilizing algorithm/system makes the obvious assumption that it is executed. This assumption is not simple to achieve since both the microprocessor and the operating system should be self-stabilizing, ensuring that eventually the (self-stabilizing) applications/ programs are executed. An elegant composition technique of self-stabilizing algorithms is used to show that once the underlying microprocessor stabilizes the self-stabilizing operating system (which can be started in any arbitrary state) stabilizes, then the self-stabilizing applications that implement the algorithms stabilize. This work considers the important layer of the operating system.

The work presents several design solutions on a scale ranging from a black-box solution to several tailored solutions. The proofs and prototypes show that it is possible to design a self-stabilizing operating system kernel. A provable self-stabilizing operating system kernel can also form the basis for a protected host system against possible malicious programs. Thus, this work provides concepts for building a system that can automatically recover from an arbitrary state including even one in which a Byzantine execution of one or more programs repeatedly attempts to corrupt the system state.

Chapter 1

Introduction

This work presents several approaches for designing self-stabilizing operating systems. The first approach is based on periodical automatic reinstalling of the operating system and restart. A variation of this approach, reinstalls the executable portion of the operating system and uses predicates on the operating system state (content of variables) to ensure that the operating system does not diverge from its specifications. Then the work details the building blocks of a tailored micro self-stabilizing operating system. Lastly, the study demonstrates the way a provable self-stabilizing operating system kernel can form the basis for a protected host system against possible malicious programs. Prototypes using the Intel Pentium processor were composed.

In his pioneering work on the "The" system [40], Edgar W. Dijkstra developed what might be considered the first operating system. He also introduced the concept of self-stabilization in [38]. This current work is a small step towards uniting these two concepts by implementing a self-stabilizing operating system. This effort should lead to robust systems and hence better service for us all.

1.1 Motivation

The robustness of an operating system is, in some cases, more important than its performance [27, 52, 58, 69, 78, 79, 99, 112]. The experience with existing operating systems, and in fact with every large on-going software package, is that it almost has its own independent behavior. The behavior is tuned up and modified by system administrators who constantly and continuously try to monitor it. The system is usually too complicated to monitor. The system administrators use human behavior and character terms, as if the system is an entity with its own will, to refer to its input output scenarios. The importance of a design that is based on well understood theoretical paradigms, and give us control over the resulting system cannot be exaggerated. In particular in

the case of the operating system, robustness is a must, as the operating system forms a basic infrastructure in almost every computing system (Here we concentrate on robustness in regard to operating systems since correctness is seldom a requirement for the common operating systems. Later on the in this work, we actually require both).

Designing a robust operating system is a complicated and challenging task. The system designer makes several probabilistic assumptions that may not hold if an execution is long enough. For example, soft errors [81, 97, 105] may cause an arbitrary change in memory bits that the error correcting schemes used will not identify. Another example, is that the communication between the system components can be made reliable, say by the use of error correcting codes. However, this assumption is also based on probability (where the life length of the system is a parameter). Once the probabilistic assumptions do not hold the designer can no longer guarantee much.

This work proposes several approaches for designing an automatic recovering operating system that are based on the well defined and well understood self-stabilization paradigm [38, 41, 124]. Roughly speaking, a system is *self-stabilizing* if it can be started in any possible state and converge to a desired behavior. A *state* of a system is an assignment of arbitrary values to the system's variables.

A self-stabilizing algorithm/system makes the obvious assumption that it is executed. This assumption is not simple to achieve since both the microprocessor and the operating system should be self-stabilizing, ensuring that eventually the (self-stabilizing) application programs are executed. An elegant composition technique of self-stabilizing algorithms [41, 42] is used here to show that once the underlying microprocessor stabilizes [43] the self-stabilizing operating system (which can be started in an arbitrary state) stabilizes, and then the self-stabilizing algorithms that implement the applications, stabilize. This work is part of an effort to create a stack of tools that provide a software platform for executing self-stabilizing programs [22, 23]. Here we consider the important layer of the operating system. Operating systems are essential parts of most computer systems [130, 138]. The operating system manages the hardware resources, and forms an abstract (virtual) machine that is convenient to program by higher level applications developers.

An operating system kernel usually contains the basic mechanisms for managing hardware resources. The classical von-Neumann machine architecture includes a processor, a memory device and external I/O devices. Thus, the core building blocks of an operating system kernel are addressed.

Operating systems are not self-stabilizing. The operating system is typically a large software constantly executed by a processor. Fault free software is a hard task to achieve (see, e.g., [24]). When the operating system is designed

for a specialized restricted task, such as (minimal configuration of) the TinyOS [67], formal methods of verification may assist in achieving fault free software. Still, the resulting system may fail due to a transient fault (e.g., a soft-error). Therefore a self-stabilizing approach is a must in such basic and on-going components as an operating systems. Apparently the current design of operating systems does not take into account the automatic recovery property of the system as a basic requirement. For example, there are processors (e.g., Intel's Pentium) which cannot support an implementation of a self-stabilizing operating system. For example, these processors are designed to support external interrupts. One class of these interrupts are the Non-Maskable Interrupts (NMI). While the operating system is handling an NMI, the processor is not reacting to additional interrupts. To enable additional interrupts the `iret` machine command must be executed [75]. Self-stabilizing systems must be able to start from any initial state in particular, a state in which interrupts are masked, and therefore should either repeatedly execute `irets` or should not use interrupts. It turned out that the Non-Maskable Interrupt is in fact *maskable*. The NMI may be masked by I/O memory instructions (e.g., [111]). Also when the system changes mode to the system management mode (say, due to transient faults) the NMIs are actually disabled [75] (Vol. 3, Sec. 13.8).

The self-stabilizing approach for modeling faults is orthogonal to the Byzantine faults model [34, 46, 83], both approaches can in fact be combined [46]. While in the Byzantine model faulty processors (nodes) may exhibit malicious behavior (representing a worst case change in the program the processor executes) the fault model used for self-stabilizing system assumes that (at least a portion of) the processes in the system execute a correct code (e.g., [46]). The requirement for code correctness (of at least two thirds, or so, of the processes) is obvious, even in the case of Byzantine faults [83]. If all the processes are Byzantine then the system can exhibit any behavior. The requirement concerning the fault-free programs can be achieved by designing the system to repeatedly access a fixed read only memory device that reloads the executable code from, say, a compact disk.

The NMI example is not based on corrupting the code of the algorithm but only on changing its (soft) state, namely altering the content of the variables. An additional prominent example of the lack of stabilization with regard to the processor/operating system interaction design, is related to interrupt handling. Usually, an operating system maintains an interrupt descriptor table (IDT). The IDT contains pointers to the different operating system routines called upon interrupts, such as the timer/clock interrupt. The Pentium processor has a register pointing to this table (IDTR). A transient fault that causes a value change of this register may disable the entire interrupt capability, and even cause the processor to execute a useless infinite loop. A similar scenario is possible when the interrupt table itself is corrupted.

1.2 Thesis and Contribution

Approaches for self-stabilizing operating systems. One approach in designing a self-stabilizing operating system is to consider an existing operating system (e.g., Microsoft Windows, Linux) as a black-box and add components to monitor its activity and take actions accordingly, such that automatic recovery is achieved. We will call this approach the black-box based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We will call this approach the tailored solution approach. The work presents several design solutions in the scale of the black-box to the tailored solutions.

Blackbox. The first simplest technique proposed here for the automatic recovery of an operating system is based on repeatedly reinstalling the operating system and then re-executing. The second technique is to repeatedly reinstall only the executable portion, monitoring the state of the operating system and assigning a legitimate state whenever required. In the blackbox approach we assume that an operating system code is correct, but it may reach a state that was not expected, namely corrupted variables values (due to memory leaks, unexpected IO sequence from the environment, etc.). A primitive solution for achieving fault-free programs is to design the system to repeatedly access a fixed read only memory device (e.g., compact disk) and reload the executable code from it. The reloading procedure is hardwired in ROM and is operated using watchdog and NMI (Non-Maskable Interrupt) mechanisms (which are described in Chapter 3). The black-box approach is discussed in Chapter 4.

Tailored Approach. An operating system kernel usually contains basic mechanisms for managing hardware resources. The classical von-Neumann machine includes a processor, a memory device and external I/O devices. The tailored operating system is built (like many other systems) as a kernel that manages these three main resources. The usual efficiency concerns which operating systems must address, are augmented with stabilization requirements. In Chapter 5 scheduling issues are investigated. In Chapter 6 memory management schemes are addressed, and in Chapter 7 device drivers are handled.

- **Process Scheduling.** The system is composed of various processes which are executing each in turn. The process loading, executing and scheduling part of the operating system usually forms the lowest and the most basic level. Two main requirements of the scheduler are fairness and stabilization preservation. Fairness means that in every infinite execution every running process is guaranteed to get a chance to run. Stabilization preservation means ensuring that the scheduler preserves the self-stabilization property of a process in spite of

the fact that other processes are executed as well (e.g., the scheduler ensures that one process will not corrupt the variables of another process).

The scheduler is the key to executing all other processes, therefore, its correct starting and execution must be guaranteed. The watchdog and non-maskable interrupts mechanisms ensure periodically executing the scheduler. Additionally, the state of the scheduler must be validated for correctness. The scheduler uses a process table for scheduling management. This information must be correct in an on-going execution, and must adapt to different scenarios, e.g., starting applications to handle external inputs. Stabilization preservation is achieved by means of monitoring processes and program code restrictions.

• **Memory Management.** We deal with two important requirements to the tasks of memory management. The first requirement is the *eventual memory hierarchy consistency*. Memory hierarchies and caching are key ideas in memory management. The memory manger must provide eventual consistency of the various memory levels. The second requirement is the *stabilization preservation* requirement. It means that stabilization proof for a single process p is automatically carried to the case of multiprocessing in spite the fact that context switches occur and the fact that the memory is actually shared. Namely, the actions of other processes will not damage the stabilization property of the process p.

The work suggests three basic design solutions that follow the evolution of memory management techniques. The first approach allocates the entire available memory to the running process, thus, ensuring exclusion of memory access. Since each process switch requires expensive disk operations, this method is inefficient. The second solution partitions the memory among several running processes and exclusive access is achieved through segmentation and stabilization preservation of the segment partitioning algorithm. Both solutions constrain program to reference addresses in the physical memory only (or even in the partition size) and allow static use of memory only. The last solution uses lease-based dynamic schemes, in which the application must renew memory leases in order to ensure the correct operation of a self-stabilizing garbage collector. The dynamic memory manager repeatedly checks for memory portions allocated to a process for which the lease expired, and returns every such memory portion to the available memory pool for reallocation.

• **I/O Device Drivers.** Device drivers are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing consistent and more abstract interface for other programs and the hardware device resources (and some-

times they also add extra services not provided by the hardware devices). Device drivers are known to be a major cause of operating system failures [136].

Here we define two requirements which should be satisfied in order for the protocol between the operating system and an I/O device to be self-stabilizing. The first requirement (the ping-pong requirement) states that in an infinite system execution, in which there are infinitely many I/O requests, the OS driver and the device controller are infinitely often exchanging requests and replies. The second requirement is about progress and it states that eventually every I/O request is executed completely and correctly according to some protocol specification (e.g., the ATA protocol for storage devices). A device driver and device controller can be viewed as a master and a slave working together according to some protocol to achieve their mission. Thus, the device driver acting as a master can check that the slave is following, e.g. the ATA protocol, correctly.

Two solutions are suggested. In the first solution the device controller is not required to be self-stabilizing, and the device driver leases some (usually enough) time to the device controller to complete its tasks. The second solution relaxes the timing constraints by assuming that the device controller itself is also self-stabilizing. Therefore, we only need to guarantee that the execution is carried out by both parties according to the protocol. This is achieved by the device driver performing consistency checks according to its current state.

These tailored approaches are then augmented from a security point of view by introducing stabilizing hosts.

- **Stabilizing Trust and reputation for Self-Stabilizing Efficient Hosts in Spite of Byzantine Guests.** A provable self-stabilizing operating system kernel can be leveraged to protect a host system against possible malicious programs. The work provides concepts for building a system that can automatically recover from an arbitrary state including even one in which a Byzantine execution of one or more programs repeatedly attempts to corrupt the system state. Preservation of a guest execution is guaranteed as long as the guest respects a predefined contract, while efficiency is improved by using stabilizing reputation. The provable self-stabilizing host operating system implementation is augmented with a contract-enforcement framework example. These ideas are presented in Chapter 8.

Tailored OS Implementations [132]. Prototype implementations for the various parts presented above, using the Intel Pentium processor architecture (also known as IA32)[75] were composed. Each part mentioned above is demonstrated with an implementation of the mechanisms for satisfying the needed requirements. Detailed proofs of each mechanism's correctness is provided. The implementation is written directly in Assembly language, using the pro-

cessor opcodes (we have used the NASM open-source assembler [107]). The methodology used here for building such critical systems is to examine, with extra care, every instruction while assuming an arbitrary initial state. This is achieved by writing the code directly according to the machine semantics (not relying on current compilers to preserve our requirements), together with line by line examination. This style is sometimes tedious, but is essential to demonstrate the way one should ensure the correctness of a program from any arbitrary initial state. Such a method is especially important when dealing with such a basic component as an operating system kernel. Higher level components and applications can then be composed in ways discussed in [24]. The Pentium was chosen as a specific example, since it is a widely used and available processor/architecture. Thus, similar methods can be applied to other processors as well. This work's proofs and prototypes show that it is possible to design a self-stabilizing operating system kernel.

Main Contributions. This thesis describes a self-contained suit for achieving practical self-stabilizing systems. In particular:

- Delivering the first self-stabilizing operating system.
- Demonstrating the non-stability of current widely used systems and contributing to the ongoing research on system robustness.
- Stating the additional requirements that various parts of an operating system kernel should satisfy.
- Presenting new self-stabilizing mechanisms, including resource leasing and contact enforcement by trust and reputation.
- Providing provable implementations, which form a basis infrastructure for practical self-stabilizing systems, by which robust and dependable systems can be achieved.

1.3 Work Organization.

The rest of this work is organized as follows. Chapter 2 provides background materials and related work. Chapter 3 details the models and settings for the rest of the work. We starts with the blackbox solutions in Chapter 4. Then Chapters 5, 6 and 7 describe the tailored solution according to the main kernel components. Demonstration implementations appear in the appendix of these chapters (6.7, 7.6). In Chapter 8 protection against Byzantine programs is discussed. Finally, we summarize the thesis and note concluding remarks including performance considerations in Chapter 9.

The blackbox and scheduling parts appeared in the proceedings of the 15th International Conference on Database and Expert Systems Applications (DEXA), the 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS'04) [47]. The work on memory management appeared in the proceedings of the 7th Symposium on Self Stabilizing Systems (SSS'05), and also in the Journal of Aerospace Computing, Information, and Communication (JACIC 2006) [48]. Work on I/O device drivers appeared in the proceedings of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06) [49] , and the stabilizing host against Byzantine will appear in SSS'07 [50]. The work was also presented as part of a stack of self-stabilizing enabling tools at the Second International Conference on Availability, Reliability and Security, Workshop on Foundations of Fault-tolerant Distributed Computing (FOFDC 2007) [23] and also published at the EAT-CS journal [22]. Additionally, this work was presented as a poster in the Symposium on Operating Systems (SOSP) 2005, and in a PhD track of several conferences. Three of the papers are to date in a journal review process.

Chapter 2

Related Work

2.1 Historical Brief Review

Operating systems evolved through continuous effort to solve basic problems of computer science like parallelism and synchronization (e.g., [37]). Grouping those solution into packages together with computer architecture development gradually formed today's operating systems. For example, see one of Dijkstra's late manuscripts [40] which describes in retrospect the designing of what by now would be called an operating system and its interrupt mechanism.

While there might not be general agreement about the range of problems that operating systems deal with (e.g., [130] vs. [139]). Some researches are trying to give a definition of an operating system which is consistent with their work. In [54] it is suggested to define an operating system as any piece of software that an application can neither change nor circumvent. "What the operating system should do is what no other piece of software can do, i.e., safely multiplex physical resources". During an operating system development project [86], the developers failed to realize that a kernel is not the same thing as an operating system, and gave other examples too. Others, predict that operating systems will be tailored to specific needs (no "one size fits all"), and in order to adapt to many applications, an operating system should focus only on resource management [147]. Some claim that operating systems should provide fault-tolerance abstraction and mask failures for users [95].

Building an operating system is usually a complex procedure and sometimes requires compromising between contradicting objectives [129]. For example, referring to the Minix operating system, [139] states that in some situations, "the operating system may have no choice but to print an error message and terminate." or "just to ignore the (deadlock) problem. We are faced with an unpleasant trade-off between convenience and correctness". Thus, a lot of research is focused towards robust, dependable and flexible operating

systems.

2.2 Robust System Efforts

As noted already in the abstract, computer system robustness remains an important and un-solved problem, e.g. [112]. Since early days of modern computing it was clear that these systems should be built to cope with errors. For example, in 1956 von-Neumann introduced handling faults and the notion of gates that may fail to compute correct output [145]. Robustness is defined by [103] as the ability of (software) systems to react appropriately to abnormal conditions. It is a complementary property to correctness, and it characterizes what happens outside of the system specification. Recently, hardware manufacturers are classifying robustness features around reliability, availability and serviceability (RAS) [74]. In the presence of faults, a reliable system takes correcting actions and does not produce wrong output. Availability is the percentage of total operating time, that a system spends in providing a service. It is common to be measured in MTBF (Mean Time Between Failures). Serviceability tries to ease system diagnostics in order to detect faults early. In [8] there are definitions for a *fault* which is a flow in software or hardware. When a fault is activated, it may corrupt the system, leading to an *error*. If an error causes an incorrect behavior, a *failure* occurs. Here we are mostly concerned with *transient* faults which occur irregularly, e.g. by unexpected interleaving of inputs, by soft-errors, temporal unavailability of resources and alike. These faults are very difficult to avoid and tolerate, however their existence forces us to consider them in any combination with other faults such as component crashes or Byzantine faults.

2.3 Hardware Support

As mentioned, computer architecture was developed hand in hand with operating system needs. Examples are interrupts [40] and mechanisms like protection [84, 120, 121], TLB and MMU [75]. Some operating systems used capabilities supported by hardware [106], while other restrict the machine instruction set [55].
The basis for a self-stabilizing processor which is a must for a self-stabilizing operating system is presented in [43] and [44].
The use of a watchdog device is found in previous operating systems, e.g., for tiny operating system stability [141]. Detailed discussion about watchdog use for error detection appears in [96]. Usage of ROM residing procedures for monitors can be found in [27] and [117]. A combination of a watchdog and non-maskable-interrupts for saving the system state, analyzing problems and re-

setting the machine in case of severe situations is suggested in [7]. A commodity machine and operating system (MS-Windows) and supporting hardware architecture is suggested in [103], where interrupts are translated into non-maskable-interrupts. When an interrupt occurs, a small realtime operating system (VxWorks) is activated which responds to the interrupt, thus adding realtime capabilities to the operating system. The use of the Harvard model ([144]) for an operating system that keeps all the code in flash memory is mentioned in [67] and [91]. Many approached were introduced in order to prevent and tolerate faults. Here we mainly review works which targeted operating systems.

2.4 Minimalistic Approaches

There are a few operating systems, like TinyOS [67], which were designed to be small, usually due to small memory considerations. However, due to their small size, they can provide a framework for correctness proof (At least in their minimal configuration). In fact, the following is stated in [54]: “As a result, the operating system can be small and readily understood: both of these properties aid correctness”. Another example is [117] which suggests an operating system that can be configured to occupy only a few hundred bytes of memory. The importance of a small size operating system is also addressed in [86] and [123] where it is argued that systems, including operating systems, should be minimal since “for any function you can think of, at least some applications will find that by necessity they must implement the function themselves in order to meet correctly their own requirements” (- “End-to-end argument”).

Micro-kernel architectures are popular in some operating systems as means for increasing fault-tolerance [61]. This architecture is based on supplying a minimal kernel while moving most operating system services to run as user processes. Minix simplification [139] is gained through a kernel which mainly transfers messages between other modules. Another similar architecture is Exokernel which removes everything from the operating system except for resource multiplexing [55].

Operating systems such as Linux and Microsoft Windows support user interface in hundreds of different languages, have many drivers and other facilities. The need for such a variety, does not enable minimalism in order to achieve robustness [143].

The monitor primitive [68] was developed in the early days of operating systems, as a means for simplifying the operating system task of resource allocation. Some interesting indications why current system fail, mainly due to complexity appear in [32]. In [32] it is concluded that there is a need for simplicity and elegance in order to achieve reliable systems.

2.5 Fault Tolerance Mechanisms

Some operating systems incorporate mechanisms like **checkpointing** and **replication**, found in databases and distributed systems, others add **monitoring** layers and automatic restart options. Motivation for checkpointing in operating systems that is based on the fact that checkpointing is done anyhow during process switching in the most efficient way, appears in [4].
Microsoft Windows[101] (starting from the XP version) and EROS (Extremely Reliable Operating System)[87] use checkpointing to gain fault-tolerance. Some systems [7, 103] supply a monitoring layer for common operating systems like Linux and Windows. The ChorousOS operating system and its successor C5, monitor all (kernel and user) activity through logging in special buffers and include an API for that purpose [7, 65]. They also contain rich restart options (more for availability than reliability) and the notion of restart group for a set of components that are assumed to fail together. There is also a kernel restart group, which is the root of the dependency graph [1, 77].
An architecture with tools for monitoring legacy applications, collecting crash data and detecting abnormal behavior is suggested in [78]. Suggestion to add monitoring layer with Bayesian techniques appear in [98]. Replication and checkpointing techniques for achieving transparent fault tolerance in Unix is suggested in [21]. Similarly, [70] adds fault-tolerant services to Windows NT through replication and checkpointing. Tandem computer systems (now HP non-stop) were built as a combination of hardware with replication and a fault-tolerant kernel, built around isolated processes and messages [15].

2.6 Fault Avoidance

By using separation techniques and even design methods of software engineering, e.g. [99, 100], a system can be constructed in ways that eliminate bugs or contain them in their own modules. Dijkstra initially influenced this area (as well) by formulating design principles like the layered approach or his famous “Goto considered harmful” sentence. Another direction is using type safety and model checking. In Coyotos [126], the whole kernel, is written in a typed language as a stage towards achieving formal correctness. Static analysis of device drivers’ code appears in [11, 12]. Recently these methods were augmented with a general tool for termination checking [31]. In [12] it is claimed that kernel APIs are usually too complex, so there is a great chance for coding bugs. They categorize bugs in order to find them automatically. Many others (e.g., [30]) use code analysis to find kernel bugs in general. The Singularity project [72, 73] emphasizes system dependability. Among other mechanisms they rely on a verification process during compilation and loading. They also use software separation between processes to minimize pos-

sible harm, and rely on a typed language to restrict program's abilities [71]. To prevent malicious code behavior, runtime code changes are restricted by eliminating language features such as reflection. In Chapter 8 we expand the review on separation techniques.

2.7 Other Related Research

The widely acceptable separation of kernel from user modules enhances operating system robustness. Simplifying construction of research operating systems through extensive modularity of different operating system components is the goal of [57]. Likewise, Choices is an experimental research operating system built around an object-oriented framework for rapid prototyping [28].
Realtime operating systems were developed over the years with various fault tolerance mechanisms as mentioned above. Qnx [115] is one of the leaders with a micro-kernel architecture and restart mechanisms. Other academic projects are eCOS [117] and ChorusOS [7]. In [122] Minix is enhanced with additional scheduling algorithms, but no details for the techniques that make it "realtime" are given.
An architecture that isolates the operating system from driver failures by restricting driver access to kernel data structures, and monitors them for reliability purposes is described in [136]. An operating system which uses statistics tools to watch for various errors is suggested in [125]. In [125] compiler support for eliminating buggy program paths is presented. The system adapts through activity analysis and simulation of various competing policies. K42 was an IBM open source operating system project (part of the autonomic computing initiative) that had strong monitoring and tracing capabilities together with hot-swapping [5]. A general algorithm/mechanism that affects the system based on continuous feedback is described in [51]. In KeyKOS [87] all resources are given through persistent capabilities (see also [36] and [92]). They claim that capabilities may serve not only for external security, but also against buggy programs [87]. A self-stabilizing distributed file system for Linux, based on replication is presented in [45]. The use of a sandbox is suggested in [146]. The design of the sandbox has similarities with ours, for example, the address validation procedure and the use of read-only code (in [146] a RISC architecture is used).

2.8 Summary

Extensive theoretical research has been done towards self-stabilizing systems [41] and recovery-oriented /autonomic-computing/ self-repair/disaster-

recovery /reliability- availability- serviceability, see [24, 27, 35, 43, 58, 79, 112] and the references therein. Some systems are built to cope with severe fault combinations such as [27, 45].

However, none of the above suggests a design for the core of an operating system that is self-stabilizing and can withstand any combination of transient-faults, which is the goal of this work.

Chapter 3

System Models and Assumptions

3.1 The System Model

This chapter defines an abstract model for the operating system and its interaction with the microprocessor/hardware on one hand and with the applications/users on the other hand. We start in defining the model for a processor.

The processor. The Intel Pentium processor [75] was chosen as a specific example to refer to when presenting our requirements and design approaches for self-stabilizing operating systems. In the sequel the notation [75] {volume/ chapter.paragraph} is used to refer to a particular item of the Pentium manual. For simplicity we only consider the *real-addressing* mode of operation of the Pentium processor which omits protection mechanisms [75]{1/3.3} (the results presented here can be applied, with a few modifications, to the *protected mode* as well).

The *processor* (or microprocessor) is defined by an operation manual, e.g., [75]. The microprocessor *state* is defined by the contents of its internal memory and the values of its input/output pins, $\langle registers, i/o\, pins \rangle$.

The *registers* (see [75]{1/3.2, 1/3.4}) include the *program counter* (`pc`) or instruction pointer (`ip`) register. In [75] this register is actually a combination of several registers according to the processor addressing mode. To simplify the discussion we assume there is a single register containing the physical address of the next operation to be performed. The registers also include the *processor status word* (`psw`) or `flag` register, which determines the current mode of operation. In particular, the `psw` contains a bit indicating whether interrupts are enabled.

The *general purpose registers* are used for arithmetic-logic operations, memory addressing and indexing operations. The *stack registers* mark the base and head of a stack in the memory. For this model we omit other registers which typical processors have, such as the special arithmetic registers. Also we do not

deal with internal control and microcode registers that cannot be controlled by the operating system programmer.

A *clock tick* (or pulse) triggers the microprocessor to *execute a processor step*, where the inputs and the current state of the processor are used to define the next processor state and the outputs. The *inputs and outputs* of the processor are the values of its I/O *pins* whenever a clock tick occur. The processor uses the I/O pins values to communicate with other devices, mainly with the memory via its data lines [75]{1/13}. The *interrupt pin* which is connected to external I/O devices, is used to signal the processor for (urgent) service requests. When the pin is set and the interrupt flag bit (`if`) is cleared in the PSW, the processor saves the program counter and PSW register values in the memory address pointed to by the stack registers. Then the processor loads the program counter with a value of an appropriate interrupt handling procedure memory address, causing the processor to leave the currently executed code [75]{1/6.4, 3/5}. The NMI *pin* acts as an interrupt pin, except that the NMI request is not masked by the interrupt flag. When the processor handles one NMI, other NMI's are ignored until an `iret` operation is executed [75]{3/5.7}.

A *processor step* $ps_j = (s, i, s', o)$ is a transition (triggered by a clock tick) from the processor state s to another state s', where s' and the values of the output pins of the processor, o, are determined by the current state of the processor, s, and the current inputs, i. In fact the processor can be viewed as a transition function defined by, e.g., [75]{2/3.2, 4.1}. A *processor execution* $PE = ps_1, ps_2, \cdots$ is a sequence of processor steps such that for every two successive steps in PE, $ps_j = (s, i, s', o)$ and $ps_{j+1} = (\overline{s}, i, \overline{s}', o)$ it holds that $s' = \overline{s}$.

Next we move to describe the model for the entire system.

The system. The *system* is modeled by a tuple $\langle processor, memory, I/O\ connectors\rangle$. The *memory* is the direct storage device connected to the processor. It has a linear address space which is used for accessing ROM and RAM components. The memory contains code and data of the operating system as well as of other applications. The memory also contains a place for the stack and an interrupt descriptor table (IDT) which holds addresses of interrupt handling routines which also reside in memory. The ROM part of the memory is non volatile and its content is guaranteed to remain unchanged. I/O state is the value of the pins connecting to peripheral devices. We assume that any information stored in the interface cards for these devices, is part of the memory.

We assume that in every infinite processor execution, PE, the processor executes fetch-decode-execute infinitely often. Moreover, the processor executes a fetched command according to its specification [75]{2/3.2, 4.1}, where the state of the processor when the first fetch starts is arbitrary. The assumption concerning the repeated execution of fetch-decode-execute can be achieved by techniques presented in [43] and [44].

A *system configuration* is a processor state together with the content of the

system memory. A *system execution* $E = (c_1, a_1, c_2, a_2, ...)$ is a sequence of alternating system configurations and system steps. A system step consists of a processor step together with the effect of the step on the (external) memory (and other non stateless devices, if they exist). Note that the entire execution can be defined by the first (for achieving self-stabilization usually assumed arbitrary) configuration and the external inputs at the clock ticks.

Self-stabilization. roughly speaking we would like the system to converge to a desired behavior following the occurrence of transient faults. We define the desired behavior by a set of system execution called *legal executions*. The legal execution can be syntactically defined (which is some times tedious), by defining the allowed combination of variable values in a configuration, called a *safe configuration*. The set of safe configurations is closed under system step executions and the infinite executions that starts in a safe configuration are proven to be legal executions. Once safe configurations are defined and proven safe, the system designer should prove stabilization by proving that any infinite system execution includes a safe configuration. Here we give a different implicit definition of the set of legal executions.

We define a *legal execution* to be any execution that starts in a system configuration in which the operating system is loaded properly into the memory and the program counter points to the memory location of the first operating system machine command, and during the execution the operating system carries its job exactly according to the operating system specifications (defined by e.g., a manufacturer manual). We also allow every suffix of a legal execution to be in the set of legal executions.

A *weak legal execution* is an infinite concatenation of non empty prefixes of legal executions, thereby allowing repeated restarts of the system.

We conclude the definition of a self-stabilizing operating system as follows:

• An operating system is a *self-stabilizing operating system* iff every infinite execution of the system has a suffix in the legal executions set.

• An operating system is a *weak self-stabilizing operating system* iff every infinite execution of the system has a suffix in the weak legal executions set.

3.2 Additional Necessary and Sufficient Hardware Support

We suggest augmenting the system with several devices/components to enable the system to be self-stabilizing. In particular, we suggest adding (1) a watchdog device, (2) a mechanism causing the microprocessor to leave the NMI state automatically after a predefined number of clock ticks, and (3) a read-only memory for the code of the program and the interrupt table.

A *watchdog* is a device connected to the NMI pin which is guaranteed to pe-

riodically generate a (reset) signal in a predefined time interval. Watchdog devices are standard devices used in fault-tolerant systems e.g., [27, 43]. We have to design the watchdog to be self-stabilizing as well. The watchdog state is in fact a countdown register with a maximal value equal to the desired interval time. Starting from any state of the watchdog, a signal will be triggered within the desired interval time and no premature signal will be triggered thereafter. The watchdog signals the processor through the NMI pin (in the first presented scheme of Chapter 4, it may trigger the RESET pin instead).

In order to guarantee that the processor will react to an NMI we suggest the addition of an internal countdown register or NMI *counter* as part of the microprocessor architecture (maybe part of the PSW). This NMI counter will be decremented in every clock tick until it reaches zero. Whenever an NMI is executed the NMI counter is raised to its maximal value (chosen to be a fixed value greater than the expected execution length of the NMI handler). The processor does not react to NMIs when the NMI counter does not contain zero. In addition, the `iret` operation assigns zero to the NMI counter. Thus, we guarantee that from any processor state NMI's will eventually be handled, and in addition, masking other NMI's during the period in which one NMI is handled. We say that a processor is in an NMI *state* whenever the NMI pin is set and the NMI counter contains 0, which means that the next operation that will be executed is the first operation of the NMI handler procedure[1].

A read only memory (ROM) device which is assumed to be non-volatile should be used. Specifically, the ROM will contain at least the interrupt table entry for the NMI and the NMI handler routine (which in some of our designs is an operating system stabilizer procedure).

As demonstrated in Chapter 1 the IDT which contains the NMI handler address is pointed to by the IDTR register. We assume that the IDTR register value can not be changed.

As previously mentioned, we assume that the processor is always in real mode (hardwired flag value ensure that it does not change state to protected mode). Otherwise we would have more complications to ensure that the NMI is executed as desired since in protected mode the value in the IDTR points to a descriptor and not to a plain address.

Figure 5.1(a) of Chapter 5 illustrates the legal execution of the system. The system is composed of various processes all of which execute in there turn. Additionally, there is a scheduler which is part of the NMI handler (details concerning the scheduler appear on Chapter 5). The scheduler repeatedly establishes its own consistency and also carries the process switch operation. It then validates the next process' state and sets the program counter so that the next chosen process will be executed. Due to a fault, the system may reach

[1] Note that the Pentium design has a similar mechanism that ensures that no NMI is executed immediately after an `sti` instruction [75]{2/4.1-STI}, see also [75]{2/4.1-POP}.

any possible state, as seen in Figure 5.1(b). However, due to the NMI trigger design, eventually the scheduler code will be called and will establish the required behavior.

Note that some of the required hardware is widely used in embedded solutions (e.g., platforms for the systems mentioned in Section 2.4). However, the ubiquitousness of the IA32 platform allows us to: a) easily find simple needed peripheral implementations. b) establishing a solution based on a vastly deployed platform.

3.3 The Error Model

We assume that every bit of the system's variables might change following some transient fault (e.g. soft-error). We also assume that some code portions are kept in read-only nonvolatile memories which can not be corrupted (say, by means of hardwired ROM chip) and, thus, are not part of the system's state. We remark that a corruption of the code may lead to an arbitrary (Byzantine) behavior!

Chapter 4

Blackbox Techniques

Chapter Organization. Section 4.1 describes the first simplest technique we propose for the automatic recovery of an operating system, where the operating system is reinstalled and then is re-executed. Section 4.2 adds details for reinstall of the executable portion, monitoring the state of the operating system and assigning a legitimate state whenever required.

4.1 Periodical Reinstall and Restart

In this section we present two design options for augmenting existing operating systems with an additional self-stabilizing layer that will periodically reinstall the executable code of the given operating system, and then either start executing from the predefined initial command statement, or from the position of the program counter prior to the reinstall. Both approaches have the flavor of a *software rejuvenation* technique [69], but do not result in a purely self-stabilizing operating system. In the first design scheme the system is (periodically) reinitialized even when it is operating well. In the second design, even though the execution continues from the same executable position using the (soft) state prior to the executable rejuvenation, the soft state variables may be inconsistent, and therefore the system as a whole will not be in a consistent state. Still, both schemes lead to more sophisticated solutions.

Periodic re-install and *start* execute. The additional watchdog device (see Section 3.2) periodically (when the period is long enough for the system to operate, say every few days) resets the processor and causes it to enter a state from which the processor reinstalls the entire operating system and starts executing it from the beginning.

Periodic re-install and *continue* execute. Similar to the former options, only this time we use the interrupt semantics of the processor to ensure that after

re-installation the system continues it's execution from where it stopped.

Implementation issues. Right after a computer is booted, the core of an operating system is loaded to the computer's Random Access Memory (RAM) which we will simply call *memory*. This is done by procedures residing in non-volatile Read-Only-Memory (ROM) such as EPROM (Erasable Programmable ROM) which is also called Basic-Input-Output-System (BIOS) or more broadly firmware. We would like to guarantee that the code of the operating system is correctly loaded. Assuming that the operating system code itself is self-stabilizing, it might happen that the memory holding the operating system code will be corrupted, (e.g., due to soft errors) leading to a situation in which the operating system does not function as desired and obviously will not converge to a valid state (will not stabilize). In order to cope with code corruptions we use a BIOS flavor procedure which will reside in ROM. This procedure will periodically reinstall the operating system from a (CD) ROM image (for example) and then restart the execution of the operating system.

```
; copy OS image
1   mov ax, OS_ROM_SEGMENT
2   mov ds, ax
3   mov si, 0x00
4   mov ax, OS_SEGMENT
5   mov es, ax
6   mov di, 0x00
7   mov cx, IMAGE_SIZE
8   cld
9   rep movsb

; prepare for journey
10  mov ax, OS_SEGMENT
11  mov ss, ax
12  mov sp, 0xFFFF
13  push word DEF_FLAG ;0002H
14  push word OS_SEGMENT ;cs
15  push word 0x0 ;ip
16  iret
```

Figure 4.1: Operating System Watchdog/Reinstall Procedure

As mentioned, Intel's Pentium 4 processor was chosen for demonstrating the self-stabilizing watchdog/reinstall technique. The procedure, called the watchdog/reinstall procedure, is presented in Figure 4.1. It is written in assembly language, which is directly assembled into the processor's opcode (using the NASM open-source assembler [107]). This procedure resides in the ROM and is reached following the NMI trigger. Note that any invocation of the NMI

will be followed by executing the watchdog/reinstall procedure. A description of this procedure follows.

In lines 1-9 of Figure 4.1 the whole operating system code and data are loaded from the ROM into the RAM. This is done by assigning an index register (`ds+si`) with the fixed memory address of the source bytes in the ROM (Lines 1-3) and another index register (`es+di`) with the fixed destination RAM (Lines 4-6). In line 7 the `cx` register is assigned by the operating system code length and in line 8 the direction of the copy operation is set. Line 9 performs the actual copy process while decreasing `cx` from the fixed code length (bounded by `cx` and NMI counter size) down to 0. Lines 10-15 prepare the system configuration with the address of the operating systems first command. This includes setting the stack registers to point to the stack in memory (lines 10-12) and pushing to the stack the fixed values which form the program counter (`cs+ip`) and the `flag` register to the initial values (e.g., for the flag it is "2" according to [75]{1/3.4.3}) for running the operating system in memory (lines 13-15). Line 16 performs two assignments in one step: the assignment of the program counter by the value of the beginning of the operating system code, and secondly the assignment of the flag register value by popping the value from the created stack. Lastly, this `iret` command of Line 16 enables NMIs.

Correctness proof. We will now prove that the watchdog/reinstall procedure combined with a given operating system forms a weakly self-stabilizing operating system. Recall that a processor is in an NMI state whenever the NMI pin is set and the NMI counter contains 0, which means that the next operation that will be executed is the first operation of the NMI handler procedure.

Lemma 4.1.1. *In every infinite execution E, the program counter contains the address of the watchdog/reinstall procedures first instruction, infinitely often.*

Proof. Since the execution is infinite, it suffices to show that beginning from any state, the address of the watchdog/reinstall procedure will be put into the program counter after finitely many steps. The design of the NMI counter ensures that in each clock tick the value of the NMI counter is reduced by one until it reaches zero unless the NMI procedure is started to be executed. Thus, if no NMI procedure is started to be executed then there is an infinite suffix of the execution in which in every configuration the NMI counter is zero. The watchdog triggers NMI's infinitely often so NMI state is reached and in the next step the processor will insert the address of the watchdog/reinstall procedure into the program counter. □

Lemma 4.1.2. *In every execution E of a self-stabilizing microprocessor it holds that immediately following the starting of the execution of the watchdog/reinstall procedure, it is completely executed.*

Proof. Since we assume that the microprocessor is self-stabilizing there is a suffix of E, E', in which the microprocessor will repeatedly fetch-decode-(and properly)-execute the assembly code.

Consider the configuration c in E' following which the execution of the NMI procedure starts (in c the program counter points to the first instruction of Figure 4.1 and the NMI is set to its maximal value). Note that in the system step that is triggered by NMI, the processor is reading the watchdog/reinstall procedure address using a (hardwired internal) predefined address. The watchdog/reinstall procedure resides also in ROM.

The step that follows c starts with an NMI counter value larger than the number of operations in the NMI procedure. Thus, until the procedure completion the processor is not stopped by other NMIs or any other interrupt.

The NMI procedure has a sequential code with no loops and with only finitely many operations. Since the microprocessor meets its specifications during E' it holds that the microprocessor will execute the machine commands one after an other until completion.

Note that line 9 of Figure 4.1 should be examined further. Although written as a single operation `rep movsb` the processor is actually performing a (hardware) loop instruction for byte coping, but as mentioned in [75]{2/4.1-REP} when there are no interrupts this loop is controlled by the value of the `cx` register. The loop is executed until the register `cx` becomes zero, the value of `cx` is reduced in every loop cycle, thus the `cx` register will eventually become zero and the processor will continue to complete executing the code. Alternatively we could replace the single operation in line 9 with the required number of `movsb` operations, at the cost of lengthening the code. See also in the coming Section 5.1 for further discussion concerning the execution of loops. □

Lemma 4.1.3. *In every execution E of a self-stabilizing microprocessor it holds that the* NMI *procedure repeatedly reinstalls and restarts the operating system.*

Proof. In the code of Figure 4.1 all the registers appear for the first time as a left operand of the `mov` instruction, thus assigning the register a new value.

The watchdog/reinstall procedure reinstalls the operating system using a (non-volatile) copy that resides in ROM. The copy is performed by the following machine command sequence: (1) setting the appropriate values in the registers `ds, es, si, di` and `cx` (lines 1-7) and, (2) setting the direction flag (line 8). (3) performing the actual copy (line 9).

Since the processor is self-stabilizing there is a suffix of E in which all the above machine command are executed as described in [75]{2/3.2, 4.1}.

Finally, the watchdog/reinstall procedure uses the `iret` machine instruction to go to the beginning of the operating system code and to reset the NMI counter and the interrupt flag. The `iret` instruction enables NMIs [75]{3/5.7.1} and assigns the program counter a value from the stack. The value in the stack is set, in lines 10-15, without counting on any previous register value.

Thus the reinstall does not depend on any previous state and the system, reinstalls the operating system and starts its execution from the beginning as required. □

Theorem 4.1.4. *The system is a weakly self-stabilizing operating system.*

Proof. Lemma 4.1.1 implies that in every execution E of a self-stabilizing microprocessor, eventually the processor starts executing the NMI procedure. We claim that the configuration c in which the NMI procedure is started to be executed, is a safe configuration. By Lemma 4.1.2 a complete execution of the NMI procedure is performed following c, and by Lemma 4.1.3 the operating system is reinstalled and restarted. Thus, E has a suffix following c that belongs to the weakly legal execution set, and therefore c is safe and the system is a weakly self-stabilizing operating system. □

4.2 Reinstall Executable and Monitor State

In this section we present approaches that couple the reinstall/repair procedure with an on-line consistency check that does not trigger an execution of the reinstall/repair procedure as long as the operating system is in a consistent state.

In order to have a fully self-stabilizing operating system, we can use the operating system watchdog/reinstall procedure of Figure 4.1 with the following modifications. (1) load from ROM only the operating system code and not the data, thus refreshing the code but not resetting the operating system data structures, (2) examine whether the operating system is in a consistent state by various consistency checks, and (3) check that the return address is within the operating system code boundaries and jump over there, otherwise jump to the operating system first command.

A more sophisticated approach will use correcting actions that are less severe than reinstall and start execute. Namely, we suggest performing continuous reinstall of a monitor/restarter (see [24]) and establish consistency. In this option the stabilization procedure is more sophisticated, it periodically performs various consistency checks to the system. When inconsistencies are found automatic repair actions are taken according to the problem which arose. This approach fits more when the operating system is small, such as (minimal configuration of) the TinyOS [67], and formal predicates covering all possible inconsistencies can be developed.

4.3 Concluding Remarks

In this chapter we presented the first building blocks for self-stabilizing operating systems. The performance of an operating system running as a black-

box can be tuned according to the time period required and chosen for recovery.

We have used the above procedure (Figure 4.1, using the BOCHS [20] simulator) and changed the contents of the RAM during execution of the code, and observed that the procedure ensure stabilization, namely, the processor eventually continues to execute the correct code of the operating system.

The first two approaches presented here resemble rejuvenation techniques, which do not meet the self-stabilization requirements, and are mainly used as an introduction for the later presented fully self-stabilizing solutions. The main drawback of these methods is that the system keeps restarting (hiccups) even when the system is operating as it should, which is not always acceptable. Additionally, these techniques do not take into account corruptions to the rejuvenation system itself (thus, "not guarding the guard"). In order to achieve the goal of composing self-stabilizing operating systems, one must consider the entire state space of the systems. Thus, system fundamentals need to be carefully designed and proved. The line-by-line examination style used here, is essential for demonstrating the way one should ensure the correctness of a program from any arbitrary initial state.

Alternatively to the black-box approach, the operating system code can be "tailored" to be self-stabilizing. In this case the operating system takes care of its own consistency. This approach may obviously lead to more efficient self-stabilizing operating systems, since it allows the use of more involved techniques. The next three chapters detail this tailored approach.

Chapter 5

Process Scheduling

5.1 Introduction, Settings and Requirements

One major task of the operating system is processes scheduling. The part of the operating system which is responsible for process scheduling is the scheduler. For example, in Minix [139] the scheduler forms the lowest and basic level of the operating system kernel. The scheduler is responsible for switching the execution of the processor from one *process* to another. A *process* is modeled by a series of machine instructions and data that reside in a particular part of the memory. The scheduler uses a process table for scheduling management. The table consists of the state of the processor (internal register values) including the value of the current program counter. Whenever the scheduler is executed by the processor it may load the state of one of the processes from the table and change the value of the program counter to continue the execution of this process. The rest of the settings for this chapter are based on Chapter 3.

Requirements. There are several possible requirements that the scheduler should satisfy. Here we require:
(r1) Fairness. In every infinite execution E, (1) for every process there are infinite number of configurations in which the program counter contains an address of one of the process' instructions in memory, (2) the execution formed from taking only the configurations belonging to this process forms an execution according to the instruction specifications.

Note that fairness does not guarantee stabilization of several processes. Processes may have mutual influence through memory due to undesired assignment of values to variables read by other processes. Thus, we state the next requirement.
(r2) Stabilization preservation. In addition, we would like to ensure that the scheduler preserves the self-stabilization property of a process, namely a self-stabilizing process will be executed and reach a safe state.

Next we describe a very simple scheduler in which we require that the program of a process uses predefined (hardwired in the code) memory addresses for data and branches. Then we present a scheduler that relaxes these assumptions.

5.2 A Primitive Scheduler

The scheduler restricts the set of machine code instructions that we allow. Here we assume the Harvard model in which the code of each process is in ROM and the data is in a separate RAM area. We assume that there are no interrupts, and only branch commands (e.g. `jmp` and `jbe`) to a fixed ROM location that is within the same process commands location, and follows the current command location, are allowed. For obvious reasons, we do not allow the use of the `halt` command (and a like). Similarly, we assume that the references to RAM addresses are fixed in the code (rather than indirect addresses). To furthur simplify our claims, we also require that the code of the processes does not contain loops. This way the advancement of the program counter, clearly results in switching between processes.

The above assumptions are used to prove that eventually every self-stabilizing process eventually stabilizes (see [42] for observations concerning parallel composition).

The code of an independent self-stabilizing process is a do forever loop. We remove the loop from the code and replace it with a scheduler designed to repeatedly execute the internal commands of the do forever loop.

The idea is to write the code of N processes in the ROM one after an other and to add a `jmp` command to the first line of the ROM in every unused ROM location. This way we view the execution as an execution of a single program that consists of several sub-program that are executed infinitely in a fixed predefined order. We do not allow (global) stack operations (a separate stack for each process may be carefully used). We also assume that the program counter always holds an instruction starting address (see details in Section 5.3).

Note that the restrictions concerning the machine code are necessary since the program counter may point (due to a soft error) to some data area. The data maybe interpreted as, say, a self-loop (`jmp` machine code) or even as a `halt` instruction.

Theorem 5.2.1. *The primitive scheduler is a self-stabilizing scheduler.*

Proof. Given any self-stabilizing processor state the processor will eventually fetch one of the commands in the ROM. This command is either a command of one of the processes or a `jmp` to the first command in the ROM. By our restrictions of the choice of allowed machine instructions, the processor will

continue executing machine instructions and will reach the first instruction infinitely often.

The first process will be executed from the first instruction infinitely often. Whenever the processor reaches the last instruction of the first process' original do forever loop, the processor will execute the first instruction of the second process. Thus, the second process' first command will be executed infinitely often. Similarly, the next processes will be executed infinitely often. Therefore, the fairness requirement holds.

Each process is assumed to be self-stabilizing when executed by itself. Since every process is executing infinitely often and does not change the state of other processes (does not have access to the variables of other processes) every process eventually stabilizes, and the stabilization preservation requirement holds as well. □

5.3 A Self-Stabilizing Scheduler

In this section we further enhance the capabilities of the scheduler, allowing the scheduler to switch processes in any line of their code. In contrast to our primitive scheduler we allow fine tuned fairness among processes that have different numbers of steps. For example, a process with a thousand sequential machine code lines will not cause a delay in executing a process with only ten machine code lines. Another issue is the possibility to change the code of the processes that resides in RAM as opposed to code hardwired in ROM which we assumed in the case of the primitive scheduler.

In order to allow a (semi) dynamic set of processes we cannot assume the Harvard model. We do assume that the programmer of the processes makes sure that the branches (e.g., `jmp`, `jbe`) are always to an address within the process code location (which is in fact part of the process code correctness) and the data of each process resides in a distinct separate RAM area. When there is a mixture of data space it is possible that stabilization of each process when executed separately, may not imply stabilization when scheduled by our scheduler. The reason is mutual unplanned updates of variables.

We also restrict the set of allowed machine code instructions. We do not allow stack operations (processes may implement a stack in their own data area and access it with general registers). We assume that the processes do not generate interrupts nor exceptions. For obvious reasons, we do not allow the use of the `halt` command. Note that the Intel architecture allows variable instruction length [75]{2/3.3.1}. This may cause interpreting partial instruction as a different instruction which can lead to say, mutual jumps between such partial instructions. A solution to this problem may be achieved by padding each instruction with `nop`s up to a fixed length (an assembler post processing might be relevant here) and the restriction of the scheduler that the `ip` value holds

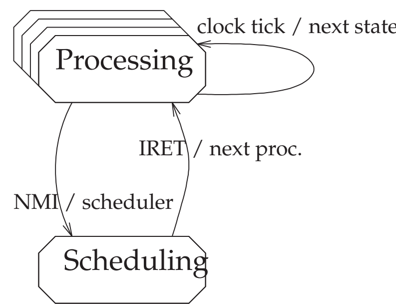
an address of an instruction start, before the execution of `iret`. (Section 4.1 did not have to address this issue, since the NMI procedure sets the program counter to a fixed correct value).

The main idea used by our scheduler is to maintain a *record* for each process of the N possible processes in a fixed location in the RAM. The record includes register values from the last state of the processor, when the process stopped being executed, namely, the values of the general purpose registers, the flag register and the program counter. The code of each process will be (repeatedly) read by the scheduler form a secondary memory device (e.g., CD-ROM) with additional information such as the length of the code. The code for reloading the processes' code can be similar to the one presented in Section 4.1, alternately, one may assume that a special process among the N processes, that resides in ROM, is responsible to refresh the code of the remaining processes.

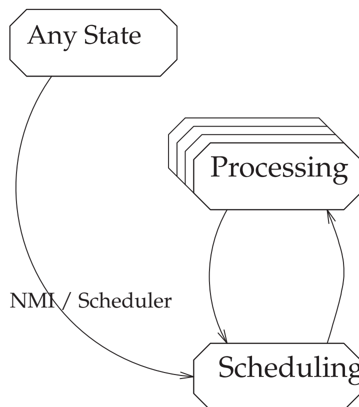
The scheduler code itself resides in ROM and is activated by the NMI as in Section 4.1. When an NMI occurs, the processor first pushes the program counter and the flag register values to the stack. The scheduler has a variable in the RAM (a part of its own state) that keeps the index of the process that is currently executed. We choose the number of bits used for the above variable to be $\lg(N)$, which implies a correct scheduler state with any arbitrary content of the variable (assuming N to be a power of 2, which can be achieved by adding some "dummy" processes).

Figure 5.1(a) illustrated the legal execution of the system. The system is composed of various processes all of which execute in there turn. Additionally, there is a scheduler which is part of the NMI handler. The scheduler repeatedly establishes its own consistency and also carries the process switch operation. It then validates the next process' state and sets the program counter so that the next chosen process will be executed. Due to a fault, the system may reach any possible state, as seen in Figure 5.1(b). However, due to the NMI trigger design, eventually the scheduler code will be called and will establish the required behavior.

Interrupts that are not NMI interrupts and exceptions (for example division by zero, caused by transient fault) are handled by default handlers that reside in the appropriate addresses in ROM (which is correctly linked by the interrupt-descriptor-table). One possibility to cope with exceptions is to use a procedure that reloads the operating system similar to the scheme presented in Section 4.1. The code of the scheduler appears in Figures 5.2 to 5.5. Roughly, the scheduler performs the following: (1) refreshes fixed addresses stored in `ss`, `sp`, `ds` and stores `ax,bx,ds` values towards sequel use (Figure 5.2 Lines 1-8), (2) saves interrupted process state in a process table (Figure 5.3 Lines 9-33), (3) increments current process variable (Figure 5.4 Lines 34-37). The (4)th and last portion deals with loading the state of the next process into the processor registers, verifying that the program counter of the loaded process is within the process RAM limits and switch to execute the next process (Figure 5.5 Lines



(a) Non-faulty execution



(b) Recovery through NMI

Figure 5.1: System Transitions

38-68). Details for each code portion follows. the reader may choose to skip the implementation details and continue with the correctness proof ahead.

The code for the refresh/store portion appears in Figure 5.2. The code starts by storing the value of `ax` in the stack to allow further execution of machine codes (in which `ax` almost always appears). The value of `ax` is stored as the immediate successor (both in terms of time and address) of the values of the program counter and flags register, that are stored during the NMI execution (line 1 of Figure 5.2). Note that the value of `ss` may be corrupted, so line 1 will result in storing the value of `ax` in an arbitrary place in memory. Fortunately, once `ss` is assigned by the fixed value `STACK_SEGMENT` (in lines 2 and 3) then it contains this fixed value throughout the execution. As mentioned above, we handle exceptions due to abnormal values of `ss`, say, during the execution of the `mov` instruction of line 1, by complete reinstallation of the operating system. Namely, the exception handler points to a fixed address in ROM in which a procedure similar to the one in Section 4.1. Lines 2 and 3 of Figure 5.2 use, the now "released", `ax` to assign `ss` with the fixed value `STACK_SEGMENT`. Line 4 assigns the fixed value `STACK_TOP` into `sp`, this is done as a preparation

```
;Verify segment and stack registers
;First save infected registers
1   mov word [ss:STACK_TOP-2], ax
2   mov ax, STACK_SEGMENT
3   mov ss, ax
4   mov sp, STACK_TOP
5   mov word [ss:STACK_TOP-4], ds
6   mov word [ss:STACK_TOP-6], bx
7   mov ax, DATA_SEGMENT
8   mov ds, ax
```

Figure 5.2: Refresh Fixed Addresses and Store `ax,bx,ds` Values Towards Sequel Use

for the `iret` instruction in Line 63 (note that this is the only use of `sp` in the code). In Lines 5 and 6 we "release" `ds` and `bx` for use in sequel instructions, we store their values in a fixed location that is related to `ss` (note that the `sp` value is not involved/changed in these instructions). In Lines 7 and 8 `ds` is assigned by a fixed value in which the index of the interrupted process and the records for each process are stored.

Figure 5.3 contains the code for storing register values of the interrupted process in memory (in the process record location). Lines 9-14 assign the interrupted process record address to the `bx` register. First the index of the interrupted process (Line 9) is copied to the `ax` register, then the value of `ax` is masked to include only $\lg(N)$ bits (Line 10) (note that for simplicity, we assume here that $N = 2^i$ for some integer i) which will be part of `al`, the least significant part of `ax`. In line 11 the address of the beginning of the memory portion in which the records of the processes are maintained is loaded into `bx`. Lines 12-14 consists of the instructions in which the address of the record of the interrupted process is calculated using the `bx` (that has been computed in line 11), the fixed size of a record that is assigned to `ah` in line 12. In line 13 the index of the interrupted process (stored in `al`) is multiplied by the value in `ah` and stored in `ax`. Note that the value of processes N and the size of the record for each process are small enough to eventually avoid overflow when multiplied (in line 13) and added (in line 14) to the value in `bx`. In lines 16-26 the values of `flag, cs, ip, ax, ds` and `bx` registers that were placed in memory (near the stack) following the `NMI` are copied to the record of the interrupted process. Each copy is performed using two instructions, the first copies the value from the fixed place in memory to `ax` and the second copies from `ax` to the record of the process, where `bx` points to the record. Lines 27-33 complete the record of the interrupted process copying the values of the other registers that form the processor state, when interrupted, to the record of the interrupted process. Note that, the register values stored in line 27-33 were not

```
9   mov word ax, [processIndex]
10  and ax, N_MASK
11  lea bx, [processTable]
12  mov ah, PROCESS_ENTRY_SIZE
13  mul ah
14  add bx, ax ;point to current process' state
15  mov ax, [ss:STACK_TOP+4] ;save flag
16  mov word [bx], ax
17  mov ax, [ss:STACK_TOP+2] ;save cs
18  mov word [bx+2], ax
19  mov ax, [ss:STACK_TOP] ;save ip
20  mov word [bx+4], ax
21  mov ax, [ss:STACK_TOP-2] ;save ax
22  mov word [bx+6], ax
23  mov ax, [ss:STACK_TOP-4] ;save ds
24  mov word [bx+8], ax
25  mov ax, [ss:STACK_TOP-6] ;save bx
26  mov word [bx+10], ax
27  mov word [bx+12], cx ;save cx
28  mov word [bx+14], dx ;save dx
29  mov word [bx+16], si ;save si
30  mov word [bx+18], di ;save di
31  mov word [bx+20], es ;save es
32  mov word [bx+22], fs ;save fs
33  mov word [bx+24], gs ;save gs
```

Figure 5.3: Save Process State

affected by the NMI nor by the scheduler activity up to this point.

```
34  mov word ax, [processIndex]
35  inc ax
36  and ax, N_MASK
37  mov word [processIndex], ax
```

Figure 5.4: Increment Process

In Figure 5.4 the scheduler increments the process index by 1 modulo N, and in this way achieves fairness among the processes (round robin). In line 34 the scheduler copies the index value of the interrupted process from a fixed place in memory. Then in line 35, the scheduler increments the process index by 1 and in line 36, performs the modulo operation by masking the result of the increment. The new calculated index is stored back to memory in line 37.

The code for loading the new process appears in Figure 5.5. In lines 38-41 the `bx` register gets the address of the record of the next process (similar to the instructions in lines 9-14). Lines 42-68 restore the next process state by copying

```
38   lea bx, [processTable]
39   mov ah, PROCESS_ENTRY_SIZE
40   mul ah
41   add bx, ax ;bx points to next process state
42   mov ax, [bx] ;restore flag
43   and ax, IP_FLAG
44   mov word [ss:STACK_TOP+4], ax
45   mov ax, [bx+2] ;restore cs
;check cs validity
46   lea si, [processLimits]
47   add si, word [processIndex]
48   add si, word [processIndex]
49   cmp ax, [si]
50   jb CS_OK
51   mov ax, [si] ;init cs
CS_OK:
52   mov word [ss:STACK_TOP+2], ax
53   mov ax, [bx+4] ;restore ip
54   and ax, IP_MASK ;validate ip
55   mov word [ss:STACK_TOP], ax
56   mov cx, word [bx+12] ;restore cx
57   mov dx, word [bx+14] ;restore dx
58   mov si, word [bx+16] ;restore si
59   mov di, word [bx+18] ;restore di
60   mov es, word [bx+20] ;restore es
61   mov fs, word [bx+22] ;restore fs
62   mov gs, word [bx+24] ;restore gs
63   mov ax, word [bx+8] ;restore ds
64   mov word [ss:STACK_TOP-2], ax
65   mov ax, word [bx+6] ;restore ax
66   mov bx, word [bx+10] ;restore bx
;finally ds
67   mov ds, word [ss:STACK_TOP-2]
;Jump to next process
68   iret
```

Figure 5.5: Load Process State

register values from the record of the process. Lines 42-44 start by copying the `flag` register value which is copied to the fixed place in stack where it will be restored by the `iret` instruction. Line 42 verifies that the loaded value is a legitimate one (although for application status bits, this is only an optimization since the application should stabilize from any state). Lines 45-52 prepare the restore of `cs`, the first half of the program counter. For simplicity, we assume that each process has a single distinct `cs` which is the most significant part of the program counter.

The value of `cs` is first copied from memory in line 45. Lines 46-51 ver-

ify that the program counter value is within the program of the process. The register `si` is assigned by the address in ROM of the table in which the fixed values of `cs` for each process are stored (adding the first address of the table `processLimits` plus twice the processIndex, since each address occupies 2 bytes in memory). In case the value of `cs` is not equal to the value pointed to by `si` (line 50), `cs` is assigned by the value pointed to by `si`, (lines 51, 52, and 68). Lines 53-55 copy the value of `ip`, the least significant part of the program counter, mask it to yield an address divisible by 16 so that it will point to the start address of the next instruction, and then copy this value to the stack. Later it is loaded to the program counter during the `iret`. We allow any value of `ip` (divisible by 16) assuming that the program of the i'th process occupies the entire memory segment that is defined by the value of `si`. (One may pad the program with `nop` instructions so that in case `ip` will hold the address of these instructions the processor will execute all the following `nops` and eventually will reach the address defined by the i'th `si` and `ip`=0). Other techniques that verify the range of `ip` in a fashion similar to `cs` validation in lines 46-51 are optional.

Lines 56-62 restore directly the values of `cx, dx, si, di, es, fs` and `gs` registers. Lines 63-66 restore the values of `ds, ax` and `bx`. Extra care is taken since `ds, ax` and `bx` registers are used for the copy process itself: `ax` as a temporary storage, `ds` and `bx` as the base and the offset to the record of a process. We first use `ax` to copy `ds` into memory lines 63-64. Then in line 64 `ax` is restored and in line 66 `bx` is restored. Finally `ds` is restored by a direct copy form the memory in line 67. At last in line 68 `iret` is performed using registers `ss` and `ip` in which we stored the (fixed) address of the stack (lines 2-4). Then the values of the program counter (stored in lines 52 and 55) and the flags register (stored in line 44) are restored. Thus, the next process is ready to be executed.

Correctness proof.

Lemma 5.3.1. *In every execution E, the code of the scheduler is started to be executed and is executed from the first instruction (line 1) to the last instruction (line 68) infinitely often.*

Proof. Since the microprocessor is assumed to be self-stabilizing there exists a suffix E' of E, in which it holds that the microprocessor will execute the machine instructions according to their defined specifications. The argument for the scheduler code being started infinitely often is exactly like the one presented previously in Lemma 4.1.1 and is based on the NMI design. The watchdog is generating a pulse infinitely often. The pulse causes the processor to load the first instruction of the fixed (in ROM) scheduler code. Also the NMI counter mechanism eventually guarantees that for some period of time after each pulse there will be no interrupts, allowing the processor to perform at

least the 68 instructions of the scheduler code. The scheduler code is sequential with no loops (the only jump address is to a fixed greater address). □

Lemma 5.3.2. *In every execution E of the scheduler each process is executed infinitely often.*

Proof. Each NMI activation causes an execution of lines 34-37 in which the process index is incremented by 1 modulo N. Then in lines 38-41 the processor state of the correspondent process is fetched from the record in RAM and copied to the actual registers in lines 42-68. The program counter is verified (lines 46-51, 54) and finally line 68 causes re-execution of that process. Based on Lemma 5.3.1 this switch will occur infinitely often thus achieving fairness. □

In case the code of the processes resides in the RAM, we may use one of the processes (maybe the first) to repeatedly reload the code of all the processes from ROM/CD-ROM. In this case, The code of the copying process itself should be in ROM. Therefore the next Lemma assumes that in every infinite execution of the system the code of each process is refreshed from ROM infinitely often.

Lemma 5.3.3. *The self-stabilizing scheduler preservers stabilization of processes.*

Proof. Each process is assumed to be self-stabilizing and has its own data area, so stabilization depends only on the correct execution of the process. Every time a process is interrupted by the scheduler the processor state is saved in lines 9-33 to RAM. When the process is rescheduled this state is reloaded in lines 38-68 to the same previous values. Thus eventually each process execution achieves the effect of a continuous execution one instruction after another, so stabilization is preserved. □

Theorem 5.3.4. *The algorithm presented in Figures 5.2, 5.3, 5.4 and 5.5 is a self stabilizing scheduler algorithm.*

Proof. From 5.3.2 fairness holds. By 5.3.3 stabilization holds, so the scheduler is self-stabilizing. □

5.4 Concluding Remarks

The scheduler component forms the lowest and most basic level of the operating system kernel. The scheduler is the key to executing all other processes, therefore its correct starting and execution must be guaranteed. This chapter demonstrated how self-stabilization methods can contribute to the robustness of this essential component.

Chapter 6

Memory Management

6.1 Introduction

This chapter presents new directions for building self-stabilizing memory managers as a component of a self-stabilizing operating system kernel. As mentioned, the classical von-Neumann machine includes a processor, a memory device and external i/o devices. In this architecture, memory management is an important task of the operating system's kernel. Our memory management components use the primitive building blocks from Chapter 5 where simple self-stabilizing process schedulers are presented.

Memory management has influenced the development of computer architecture and operating systems [16]. Various memory organization schemes and appropriate requirements have been suggested throughout the years. Here, we are adding two important requirements, named the *eventual memory hierarchy consistency* requirement and the *stabilization preservation* requirement. Since memory hierarchies and caching are key ideas in memory management, the memory manager must eventually provide consistency of the various memory levels. Additionally, once stabilization for a process is established, the fact that process and scope switching occurs and that memory is actually shared with other processes, will not damage the stabilization property of the process. These requirements are an addition to the usual efficiency concerns which operating systems must address. Usually, memory management in operating systems is handled with the assistance of quite a complex state, for example page tables. A minor fault in such a state can lead to writing wrong data onto the disk (violation of consistency) or even to corruption of some process' state (violation of preservation). Unless the system is self-stabilizing, the corruption of the tables may never be corrected and the tables' internal consistency may never be re-established.

We present three basic design solutions that, roughly speaking, follow the evolution of memory management techniques. The first approach allocates the en-

tire available memory to the running process, thus ensuring exclusion of memory access. Since each process switch requires expensive disk operations, this method is inefficient, . The second solution partitions the memory between several running processes and exclusive access is achieved through segmentation and stabilization of the segment partitioning algorithm. Both solutions constrain program referencing to addresses in the physical memory only (or even in the partition size) and allow only static use of memory.

Following this, we present lease based dynamic schemes, in which the application must renew memory leases in order to ensure the correct operation of a self-stabilizing garbage collector.

Demonstration implementations (which appear in 6.7, the appendix for this chapter) using the Intel Pentium processor architecture [75] were composed. The implementations are written in assembly language and are directly assembled into the processor's opcode (using the NASM open-source assembler [107]). The reader may choose to skip the implementation details. The Intel Pentium processor contains various mechanisms which support the robust design of memory management like segmentation, paging and ring protection. However, the complexity of the processor (partially due to previous processors' compatibility requirements) carries a risk of entering into undesirable states, and thereby causing undesirable execution. Our proof and prototype show that it is possible to design a self-stabilizing memory manager that preserves the stabilization of the running processes which is an important building block of an infrastructure for industrial self-stabilizing systems.

Previous Work. Fault tolerance properties of operating systems (e.g., [136]), including the memory management layer, were extensively studied as well. For example, in [10], important operating system memory regions are copied into a special area for fast recovery. The design of the Multics operating system pioneered issues of data protection and sharing, see [33] and [120]. The algorithms presented here enforce consistency of the data structures used for memory management. In order to use more complex data structures, the work of [66] is relevant for achieving general stabilization of data structures. [35] addresses the issue of automatic detection and error correction in common high-level operating system data structures (although, not in a self-stabilizing way). Leases, which are used here, are commonly used as a general fault-tolerant locking mechanism (see [62, 85]). In [104], leases are used to automatically remove subscriptions in publish-subscribe systems. However, none of the above suggest a design for n operating system, or, in particular, memory management that can automatically recover from an arbitrary state (that may be reached due to a combination of unexpected faults).

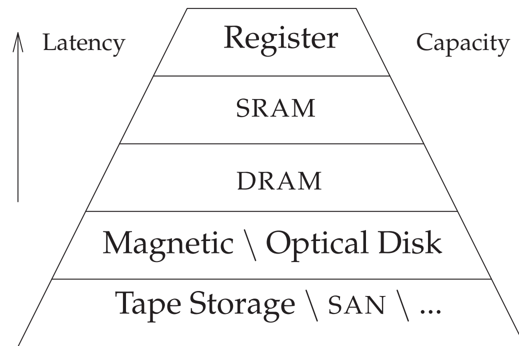


Figure 6.1: A Common Memory Hierarchy

Chapter Organization. In the next section we define the system settings and requirements. The three solutions: total swapping, fixed partition and dynamic memory allocation, are presented in Section 6.3, Section 6.4 and Section 6.5, respectively. Concluding remarks appear in Section 6.6. Code snippets with explanations appear in the appendix, Section 6.7.

6.2 System Settings, Assumptions and Requirements

We use the same settings as appeared in Chapter 3. Here we only emphasize memory related settings.

The *memory* is composed of various devices (Figure 6.1 presents some common *memory hierarchy*). Here we consider *main memory* and *secondary storage*. The main memory is composed of ROM and RAM components. Read-only parts are assumed non-volatile. The secondary storage is also organized as a combination of read-only parts, such as CD-ROM and other disks. The read-only requirement is compulsory for ensuring correctness of the code. Otherwise, the Byzantine fault model [83] must be assumed. Processor caches, at least in the current Pentium design can not be controlled directly by the operating system, and are not, therefore, considered here.

Requirements. The memory manager requirements include both the **consistency** and the **stabilization preservation** requirements:
(r1) Consistency: as the system executes, the memory manager keeps the memory hierarchy consistent (analogously to the consistency requirement for non-stabilizing operating systems). Namely we have to show that the contents of say, the main memory and the disk are kept mutually consistent.
(r2) Stabilization preservation: the fact that process and scope switching occurs, and that the memory is actually shared with other processes, will not

falsify the stabilization property of each of the processes in the system.
A *self-stabilizing memory manager* is one that ensures that every infinite execution of the system has a suffix in which both the consistency and the stabilization preservation requirements hold.

6.3 Total Swapping — One Process at a Time

In the first solution, the memory management is done by means of allocating (almost) all the available memory (RAM) to every process.

The settings for this solution are: N code portions, one for each process in the system, reside in a persistent read only secondary storage. The soft state of each process is repeatedly saved on the disk. The operating system includes a self-stabilizing scheduler (discussed in Chapter 5), which activates processes in a round robin fashion. Whenever a process is activated, the process has all the memory for its operation (except the portion used by the scheduler). The scheduler actions include saving the state of the interrupted process in the disk and loading the state of the new process whenever a process switch occurs.

The scheduler executes a process switch at fixed time intervals[1]. The processor state (register values) is saved in the stack. Note that we ensure that for every processor state, stack operations will not prevent the execution of the NMI handler and that the scheduler code will be started. This is needed since, according to the processor architecture, part of the processor state is automatically saved to the stack (during a context switch). These automatic stack operations carry the risk of unplanned exceptions. Thus, we ensure that whatever the stack state is, the handling procedure can be eventually called.

Like previous chapters, the implementation uses the Pentium processor in its real (16 bit) operation mode, thus paging and protection mechanisms that are not being used. This configuration may not be acceptable for modern desktop operating systems. Yet, it is more common in embedded systems and also serves as a simplified model for investigating the application of the self-stabilization paradigm to operating systems. The protected mode mechanisms might be used in satisfying the stabilization requirement, but once the processor exits this mode, there is no guarantee. Thus, we assume the processor's mode is hardwired during the system execution so the mode flag is not part of the system's (soft) state. For now, the disk driver operations are assumed to be atomic and stateless (achieving this abstraction is handled in Chapter 7). The obvious drawback of this solution is the need to switch the whole process state in every context switch. This might not be acceptable for all systems.

[1]Note that a clock interrupt counter may form a self-stabilizing mechanism for triggering a process switch, and that the counter upper bound is achieved regardless of what the counter value is when counting begins.

```
SWAP-PROCESS(PT, i)
1  MEMORY-SAVE-PROCESSOR-STATE(PT, i)
2  DISK-SAVE-PROCESS-STATE(i)
3  i ← (i + 1) modulo N
4  CD-ROM-LOAD-PROCESS-CODE(i)
5  DISK-LOAD-PROCESS-STATE(i)
6  MEMORY-LOAD-PROCESSOR-STATE(PT, i)
```

Figure 6.2: Total Swapping Algorithm

The scheduler algorithm which appears in Figure 6.2 carries the memory management task. The algorithm uses in its memory an array that is used for the process table denoted by PT. PT keeps the entire processor state (the register values of the processor) for each running process p_i, while i acts as a process pointer. Recall that N is the (fixed) number of processes in the system. The scheduler saves the state of the running process to the process table (line 1), and to the disk (line 2), and then increments the process counter (line 3), and loads the next process to be activated. The loading is carried by fething the process code from the read-only storage (line 4), process state from disk (line 5) and the processor state from PT in memory (line 6). The correctness of the algorithm is based on the fact that the various procedures that save and load data depend only on the value of i (that represents p_i), which by itself is bounded by the number of processes in the system. Next, we prove the algorithm correctness and then show that the implementation which follows the algorithm fulfills the requirements.

Correctness proof:

Lemma 6.3.1. *In every infinite system execution E, the program counter register contains the address of the swapping procedure's first instruction infinitely often. Additionally, the code is executed completely and every process is executed infinitely often.*

Proof. Exactly like presented previously in Lemmas 4.1.1 and 4.1.2, the processor eventually reaches an NMI state in which the NMI connector is set and the NMI counter contains 0. This means that the next operation that will be executed is the first operation of the NMI handler procedure. The system is configured to execute the scheduler as part of the NMI handler. Since the code of the scheduler is fixed in ROM, it remains unchanged. During the NMI handler execution, interrupts are not served. Additionally, the algorithm contains no loops, which enables the code to be executed completely. Since the value of i is incremented in every execution, all processes are executed infinitely often. □

Lemma 6.3.2. *In every infinite execution E, memory consistency requirement holds.*

Proof. In every context switch, the whole data portion of a process is saved to the disk and is reloaded when needed again. The addresses used for these transfers are calculated every time and are based solely on the process number. The state of the scheduler is actually only the process pointer value which is incremented in every execution and is validated to be in the range 1 to N (the process table entries are, in effect, part of each process' state). Thus, even a corrupted value of a process pointer that may cause loading or saving the wrong data for a process, will not falsify the execution of the scheduler. Consequently, next time the effected process is loaded, the process will have the correct code (thereafter, a self-stabilizing process will converge). Based on Lemma 6.3.1, the above is true in every infinite execution. □

Lemma 6.3.3. *In every infinite execution E, stabilization preservation eventually holds.*

Proof. Since the entire available main memory is allocated to each running process, the processes are effectively separated from one another. Thus, the execution of one process cannot alter the state of another process. □

Corollary 6.3.4. *The total swapping memory manager is self-stabilizing.*

Details of the implementation for this solution appear in Section 6.7.1.

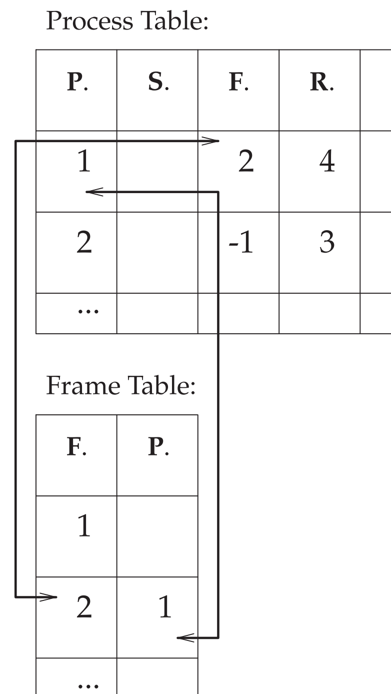


Figure 6.3: Fixed Partition Consistency Check

6.4 Fixed Partition — Multiple Residing Processes

In this section we follow a better memory utilization which allows the partitioning of memory between several processes. This reduces the number of accesses to disk, thereby improving system performance. Still, when one partition is free, the processes in other partitions can not use this free memory. So, although the second design does not require the system to repeatedly transfer the entire data between memory levels, the second design still constrains the size of the applications.

The decision concerning the set of processes that should be activated depends on external environmental inputs. This is needed since the main advantage of this solution is rescheduling processes without costly disk operations. However, since a priority mechanism is not used, all memory frames are occupied if $N > M$ (M is the number of partitions), so every context switch causes costly disk operations and the main advantage is lost. The process table is a natural candidate for holding the additional activity status for each process. The entity which generates this information as input to the memory manager is responsible for the correctness and stability of this value.

The setting for this solution is that the code of N programs resides in a persistent read only secondary storage. The operating system consists of (memory

hardwired) resident NMI handler and a scheduler process. The memory for the applications is partitioned into M fixed equal length memory segments which are called frames. Thus, programs are constrained to using the size of a frame. The operating system uses a frame table FT which describes the currently residing process in each memory frame. In addition, there is a process table PT. The i'th entry of PT consists of: (a) the last processor state of p_i for uploading in case the process should be scheduled, (b) the frame number (address in RAM) used by p_i (NIL if not present), (c) refresh down counter. When the value of the counter is zero and p_i is rescheduled, the code of p_i is reloaded from the CD-ROM in order to make sure it is not corrupted. (d) An active bit that is externally defined and flags the operating system whether p_i should be active or not. The remaining state of the processes is kept on a disk. The locations on disk and CD-ROM are calculated from the process identifier i.

Upon the periodic NMI trigger, the processor execution context (register values) is saved to the stack and the execution of the scheduler code is initiated. The scheduler saves the processor state of the interrupted process to PT, selects the next ready process, and then carries out the memory management actions necessary for executing this process. The pseudo code for the algorithm appears in Figure 6.4. In case the next process is not present in memory or if there is an inconsistency between the process and the frame tables (line 1), a new frame is chosen (line 2) and the currently residing process is saved to the disk (line 3). The refresh counter is decreased for every activation of a process (line 4). In case this value equals zero (line 5), the new process' code is loaded from the CD-ROM (line 6).

The algorithm `Find-Frame` searches the frame table for a free frame. In case all frames are used, a particular frame is chosen for replacement. First the frame currently pointed to by this process' entry is validated to be in range (line 1). Next a search over FT starts from the pointed frame's successor (line 2-5) until an empty frame or a frame containing a non-active process is found. The search continues until the whole table is looked up. Even if due to a fault, say an error occurs in the program counter which causes bypassing of lines 1 and 2 - lines that calculate the loop limit value, the execution will eventually bypass this loop. First, the size of the field used for storing the frame number in PT can be bounded by M. Thus, increments of nf (line 5), must reach the loop limit value. Secondly, the system is designed so that eventually an NMI will be triggered and the code will be re-executed from the first line.

The `Swap-Process` algorithm checks if there is a swapped out process due to the loading of the new one (line 1). If this is the case, it saves to disk the state of this process (line 2) and marks its frame entry in PT as NIL (line 3). The entries of FT and PT are updated with the new assignment (lines 4-5) and the state of the new process is loaded to main memory (line 6). Finally, the code refresh bit is set to one (line 7), a step which will cause the main procedure to decrement it further to zero and, thereafter, to load the new process' code.

```
SELECT-NEXT-PROCESS-AND-FRAME(PT, FT, i)
1  if frame[PT[i]] = NIL or FT[frame[PT[i]]] ≠ i
2    then nf ← FIND-FRAME(PT, FT, i)
3         SWAP-PROCESS(PT, FT, i, nf)
4  decrease refresh[PT[i]]
5  if refresh[PT[i]] = 0
6    then CD-ROM-LOAD-PROCESS-CODE(i, PT)

FIND-FRAME(PT, FT, i)
1  frame[PT[i]] ← frame[PT[i]] modulo M
2  nf ← (frame[PT[i]] + 1) modulo M
3  while nf ≠ frame[PT[i]] and FT[nf] ≠ NIL
4       and active[PT[FT[nf]]] = true
5  do nf ← (nf + 1) modulo M
6  return nf

SWAP-PROCESS(PT, FT, i, nf)
1  if FT[nf] ≠ NIL
2    then DISK-SAVE-PROCESS-STATE(FT[nf], nf)
3         frame[PT[FT[nf]]] ← NIL
4  FT[nf] ← i
5  frame[PT[i]] ← nf
6  DISK-LOAD-PROCESS-STATE(i, nf)
7  refresh[PT[i]] ← 1 ▷ Causes code to be loaded.
```

Figure 6.4: Fixed Partition Algorithm

After the execution of the above algorithm, the scheduler continues with the swap by loading the processor state of the new process from PT.

The correctness of the algorithm is based on the ongoing consistency checks of FT and PT. Figure 6.3 demonstrates the consistency check made when assigning a frame to a process. Frame 1 is assigned to p_2. Thus 1 is entered in the 2nd entry of FT. Additionally, the frame field in the entry of p_2 in PT (column marked with F) is marked with the new frame number. The arrow lines demonstrate the exclusive ownership of the selected frame for every scheduled process. Additionally, the refresh field (column marked with R) shows the refresh counter which ensures the periodical refreshing of the code for the processes. (The S column represents the processor state for each process).

Correctness proof:

Lemma 6.4.1. *In every infinite system execution E, the program counter register contains the address of the memory management procedure's first instruction infinitely often. Additionally, the code is executed from the first instruction (line 1 of Figure 6.9(a)) to the last instruction (line 42) infinitely often.*

Proof. The arguments for reaching the first instruction are essentially the same as in Lemma 6.3.1 (and 4.1.1). The procedure `Find-Frame` (of Figure 6.4) for finding a new frame for the running process is the only one containing a loop. This loop is bounded to run no more than `M` times, which happens when the whole frame table is scanned for a `NIL` value. If the code is executed without validation of the limit parameter (line 1 of `Find-Frame`) and without the advancing of nf, the new frame pointer, which limits the loop execution (line 2), the `NMI` mechanism will enforce an execution from the first line. This in turn, will be followed by the examination of the loop conditions and will ensure correct execution. □

Lemma 6.4.2. *In every infinite execution E, the memory consistency requirement holds.*

Proof. Based on Lemma 6.4.1 every process will be executed infinitely often and the memory management algorithm will be executed prior to the execution of the process. The memory manager will cause the correct code for such a process to be loaded infinitely often from the stable storage since the refresh counter is decremented infinitely often. Therefor, the refresh counter will reach the value zero, which will cause code reloading. The correctness of the reloaded code is based on the direct mapping between the process pointer i and a disk location (as in Lemma 6.3.2). Secondly, the target address in the main memory (frame) is validated each time to be exclusively owned by the running process. Suppose that due to a fault, two processes p_i and p_j are marked in the process table PT as residing in the same frame number f in the main memory. Whenever the scheduler activates the first process, say p_i, it first validates (in line 1 of Figure 6.4) that the f entry in the frame table FT is i. If not, a new frame will be selected for p_i and both PT and FT will be updated accordingly (line 4-5 of `Swap-Process`). Even if the new selected frame is still f, the frame table entry for f will now contain i. Thus, when p_j will be scheduled, the memory manager will detect that a new frame should be chosen for p_j.

□

Note that if, due to a fault, a frame is occupied by a non-active process, this frame will be considered as an empty frame, by the `Find-Frame` procedure (by the check made in line 4) and will be assigned to new requesting processes.

Lemma 6.4.3. *Stabilization preservation eventually holds.*

Proof. Each process eventually resides in the correct frame by Lemma 6.4.2. In addition, the applications must refer to main memory addresses in the frame size only (the implementation relies on the segmentation mechanism of the processor). Also, the code is fixed and does not change the content of the segment registers (note that such a restriction can be imposed by a compiler).

Additionally, the correctness of the segment register assignment is repeatedly checked by the memory manager. Thus, the processes are effectively separated one from another, and the execution of one process can not alter the state of another process. □

Corollary 6.4.4. *The fixed partition memory manager is self-stabilizing.*

Details of the implementation for this solution appear in Section 6.7.2.

We remark that the fixed partition restriction of the above solution can be relaxed. Applications can be of variable sizes The partition of the main memory is not fixed and a record of occupied space is maintained. Whenever a process is about to be scheduled, the record is searched for a big enough space and the application is loaded there. To ensure fulfillment of our requirements this record must be kept consistent with the process table. Additional care, using standard techniques, must be taken in order to address fragmentation of the main memory and in order avoid process starvation. The next section addresses variable memory sizes by means of dynamic allocations.

6.5 Dynamic Allocation

Further enhancement of memory usage would be to remove the static allocation nature of the programs and to allow them to allocate memory in a malloc/free style. Obviously, the operating system must keep track of memory usage based on some policy. To ensure that there is no memory which, due to some fault, is marked as used, when it is in fact unused, a leasing mechanism is suggested. In this mechanism applications must extend their lease from time to time. This way, memory that is not in use will eventually become free (assuming no malicious Byzantine behavior of processes). To be more precise, we would like to support a dynamic memory allocation scheme where additional memory beyond the fixed memory required for the code and the static variables may be allocated on-demand. To support the management of the additional memory allocations in a self-stabilizing fashion, a *lease* mechanism which limits the allocation of a new memory portion for the use of a process either by time, or the number of steps the process performed since the allocation, is used.

A memory manager process is responsible for allocating and for memory garbage collection. The dynamic memory manager uses bookkeeping for managing the dynamic memory allocations. Allocations are tracked using a table that holds the number of the owner process and the remaining lease period for each allocation unit. The dynamic memory manager repeatedly checks for memory portions allocated to a process for which the lease expired, and returns every such memory portion to the available memory pool for reallocation. The lease policy leaves the responsibility for refreshing the leases to the

programmer of the processes, and at the same time allows simple and stabilizing dynamic memory management. We can argue that starting in an arbitrary configuration, where the dynamic memory is allocated randomly to processes, eventually no memory will be allocated to a process which did not request memory (recently). Moreover, assuming no malicious process behavior in every infinite execution, repeated allocation requests will be infinitely often respected. Up until this solution, programs were totally ignorant of operating system services. Here the operating system exposes an application programming interface for memory requests. Thus, programs should now also deal with temporary rejections of requests while the operating system makes sure that eventually all legal requests will be respected. The algorithms described below address the issue of dynamic allocations. Other needed mechanisms, like the automatic refreshing of code, are taken from the previous solutions.

Figure 6.5 presents the algorithms which implement the interface which programs can call in order to use dynamic memory. The MM-Alloc procedure is used for requesting memory allocation. With MM-ExtendLease a lease extension is possible. The applications are restricted to using the allocated memory through a special segment selector register and the procedure MM-NextSegment is the only way of accessing the different segments allocated to an application. At last, applications can release their allocations with MM-Free. The operating system contains a specialized process called _MM-Validator [2] that validates the system's state concerning dynamic allocation. The algorithm is presented in Figure 6.6. Additionally, we use several service procedures which are presented in Figure 6.7.

Next, we describe how the algorithms work and consequently argue concerning their correctness. The MM-Alloc algorithm inputs are the number of allocation units (segments) required by the process and the expiration period needed. The expiration is the number of activations of the process for which the allocation will be valid. This number is bounded (at least) by the parameter length. After this period, the validator will reclaim those segments and mark them as free. In line 1 of the algorithm, the *dynamic selector* (which in the implementation is realized in a specific processor segment register) is checked for holding an empty address. If this is not the case, the meaning is that this process is already using dynamic memory and that the request is rejected in line 2 (for simplicity reasons we allow only one allocation at a time). In line 3 we check whether there are sufficient allocation units for this request through a global variable that holds this count. We assign the requested quantity to the requesting process with the _MM-Assign procedure which simply goes over all the segments in the segment table ST and marks the needed quantity as occupied. This procedure also updates the free segment variable (line 5), and sets the dynamic selector value with the address of one of the allocated segments

[2]The leading underscore marks a procedure internally called by the operating system

```
MM-ALLOC(quantity, expiration)
1  if seg(PT[currentProcess]) ≠ NIL
2     then return
3  if quantity <= freeSegments
4     then _MM-ASSIGN(currentProcess, quantity, expiration)
5          _MM-ENQUEUE(currentProcess, quantity, expiration)

MM-EXTENDLEASE(newExpiration)
1  s ← seg(PT[currentProcess])
2  if owner(ST[s]) = currentProcess
3     then lease(ST[s]) ← newExpiration

MM-NEXTSEGMENT()
1  currentSegment ← seg(PT[currentProcess])
2  if currentSegment ≠ NIL
3     then for  each s in {(currentSegment + 1) modulo NUM_SEG..
4               (currentSegment − 1) modulo NUM_SEG}
5           do if owner(ST[s]) = currentProcess
6                 then seg(PT[currentProcess]) ← s
7                      break

MM-FREE()
1  currentSegment ← seg(PT[currentProcess])
2  MM-NEXTSEGEMNT(currentProcess)
3  if currentSegment ≠ NIL
4     then if currentSegment = seg(PT[currentProcess])
5             then seg(PT[currentProcess]) ← NIL
6           if owner(ST[currentSegment]) = currentProcess
7             then owner(ST[currentSegment]) ← NIL
8                  freeSegments ← freeSegments + 1
```

Figure 6.5: Dynamic Allocation Services

(line 9). In case insufficient amount of segments is available, the request is queued through the procedure _MM-Enqueue which first checks that there is not already a queue entry for this process and consequently finds an empty slot to enqueue the request. The queue size is equal to the process number. Thus, exactly one slot for each process is reserved.

The MM-ExtendLease procedure carries out its task by validating that the requested segment is owned by the requesting process and enlarges the lease counter value. Again, this operation is allowed assuming there is no a malicious behavior of processes. A different approach can enable the extension just in cases when the queue is empty, thus preventing a repeated extension of a lease by a particular process. As mentioned before, a process can access

the allocated segments through a selector which it cannot change. In order to move between allocated segments, the process calls MM-NextSegment which looks in the segment table for all other segments and if another segment is also occupied by the calling process, its number is returned by the selector (line 6). The MM-Free procedure carries out its task by first updating the selector with another segment address (lines 1-2). It then checks if this selector is the only one owned by this process, which means that the selector should be cleared too (line 5). In lines 6-8, the released segment is checked for being owned by the process and is consequently marked as free. The global counter of free segments is updated respectively.

```
_MM-VALIDATION()
 1  for each p in {0..NUM_PROC − 1}
 2  do usingDynamic[p] ← false
 3  freeSegments ← 0
 4  for each s in {0..NUM_SEG − 1}
 5  do p ← owner(ST[s])
 6     if p ≠ NIL
 7       then lease(ST[s]) ← lease(ST[s]) − 1
 8            if lease(ST[s]) = 0
 9              then owner(ST[s]) ← NIL
10              else usingDynamic[p] ← true
11     if owner(ST[s]) = NIL
12       then freeSegments ← freeSegments + 1
13  for each p in {0..NUM_PROC − 1}
14  do if usingDynamic[p] = false
15       then seg(PT[p]) ← NIL
16  q ← top(Q)
17  if q ≠ NIL and quantity(q) <= freeSegments
18    then _MM-DEQUEUE()
19         _MM-ASSIGN(process(q), quantity(q), expiration(q))
```

Figure 6.6: Dynamic Allocation Validation

The validation and garbage collection algorithm (_MM-Validation) works as follows. In lines 1-2 it marks all processes as not using dynamic memory. This will allow the initialization of the dynamic selector for processes that are incorrectly marked as already using dynamic memory. Thus, subsequently such a process will be able to request (and get!) allocations. In line 3, the global counter for free segments is reset. Thus, only used segments will be counted (lines 11-12). The loop of lines 4-12 iterates over all segments in the segment table ST and decreases the lease for each of them. In case a lease reaches zero, the segment is marked as free (line 9). Otherwise, we mark the process as using dynamic memory (line 10). Lines 13-15 reset the dynamic selector (saved in the

process table PT) for processes that do not currently use dynamic memory. Then, we check the queue top and in case the first waiting process can be satisfied with the current free segments, it is deleted from the queue (line 18) and assigned with the free segments (line 19). The _MM-Dequeue procedure merely moves all the entries in the array having the queue progress one cell towards the queue top. It also marks the last entry as free.

```
_MM-ASSIGN(process, quantity, expiration)
1  for each s in {0..NUM_SEGMENTS − 1}
2  do if owner(ST[s]) = NIL
3        then owner(ST[s]) ← process
4             lease(ST[s]) ← expiration
5             freeSegments ← freeSegments − 1
6             quantity ← quantity − 1
7             if quantity = 0
8                then break
9  seg(PT[process]) ← s

_MM-ENQUEUE(process, quantity, expiration)
1  for each p in {0..NUM_PROC − 1}
2  do if process(Q[p]) = process
3        then return
4  for each p in {0..NUM_PROC − 1}
5  do if process(Q[p]) = NIL
6        then process(Q[p]) ← process
7             quantity(Q[p]) ← quantity
8             expiration(Q[p]) ← expiration
9             break

_MM-DEQUEUE()
1  for each p in {0..NUM_PROC − 2}
2  do Q[p] ← Q[p + 1]
3  q[NUM_PROC − 1] ← NIL
```

Figure 6.7: Dynamic Allocation Service Procedures

Correctness proof:

Lemma 6.5.1. *In every infinite system execution E, the program counter register contains the address of the validator procedure's first instruction, infinitely often. Additionally, the validator code is executed entirely.*

Proof. Based on Lemma 6.3.1 the scheduler schedules every process of the system infinitely often. and specifically the validator process. Note that the scheduler also checks that the program counter value is within the fixed limits of the

program size. The code of the validator process does not contain branches out of the the procedure limits. All the contained loops are **for**-loops with advancing index which is checked each time for being inside fixed limits. Since this index advances each time, every loop is guaranteed to reach the end criteria and have the program counter advance towards the procedure end. □

Lemma 6.5.2. *In every infinite system execution E, eventually no memory will be allocated to a process which did not (recently) requested it.*

Proof. The validator keeps decreasing the leases for all segments towards zero infinitely often. When reaching zero, the segment is marked as free i.e. not allocated to any process. The only way to increase the lease value is by a process requesting (or extending) memory. Thus, eventually every memory segment that is used by a process must have been recently allocated (or extended) by this process, or otherwise released. □

Lemma 6.5.3. *In every infinite system execution E, repeated allocation requests will be respected infinitely often.*

Proof. The processes themselves are self-stabilizing and do not behave in an unfair way. Thus, every process that holds dynamic memory will release it infinitely often, allowing other processes to allocate this memory. Since the queue can contain at most one request for each process and since no process can bypass another waiting process in the queue, each request placed in the queue will eventually be respected. □

Lemma 6.5.4. *In every infinite execution E, the memory consistency requirement holds.*

Proof. The segment table ST records the owning process for each segment. Even if the table is transiently corrupted, based on Lemma 6.5.2, eventually only requesting processes will be marked as using a segment. Thus, no other process is using this memory. The access to each segment is only via a segment register The value of which we assume cannot be changed by the process. This mechanism enforces the usage of this segment only by the marked process, achieving memory consistency. □

Lemma 6.5.5. *In every infinite execution E, stabilization preservation eventually holds.*

Proof. In this case as well there is a full separation of memory accesses of processes. The static areas are separated by means of the segment selectors from the previous solution while the dynamic areas are separated based on Lemma 6.5.4. Thus, every stabilizing process will stabilize in spite of the actual sharing of memory with other processes. □

Corollary 6.5.6. *The dynamic allocation memory manager is self-stabilizing.*

Note that the memory manager can protect itself from a greedy process by designing the MM-ExtendLease procedure such that extensions are allowed only when the queue is empty. This way, when there is a pending request, a process that holds memory will eventually loosen it. Thus, from any system state, eventually enough segments will be freed for the top queue process and it will, thereafter, be granted with its request. The meaning of this is that eventually every request will be respected.
Details of the implementation for this solution appear in Section 6.7.3.

6.6 Concluding Remarks

We have presented three classes of self-stabilizing memory management schemes: total swapping, fixed partition and dynamic memory allocation.

We note that the IBM OS/360 [16] used analogous versions for memory management. The PCP version loaded only one program at a time, the MFT used a fixed partition and the MVT allowed somewhat dynamic memory partition. Virtual address hardware were then built and used, e.g., in the MVS operating system.

In order to also support virtual addressing, the page tables have to be kept consistent. This will allow correct address translation made by the MMU (memory management unit). The page tables are also usually cached in a special memory (TLB). Consistency, therefor must also be examined for this memory structure. (To date, the Pentium's TLB is not accessible by the operating system).

6.7 Appendix. Implementations

6.7.1 Total Swapping Implementation

The implementation for the total swap solution appears in Figure 6.8. The code resides in ROM and is executed following an NMI trigger. The code for saving and loading the processor state and for incrementing the process pointer is an extension of the one presented in Chapter 5. Therefore equivalent parts are omitted here.

The code contains no loops, and all address calculations are based solely on the value of i which points to the relevant process. Note that all values are also recalculated for every execution. Namely, every variable's first ap-

```
..
1  mov word ax, [i]
2  and ax, N_MASK
;// over here processor state is saved
3  mov ax, PROCESS_SEGMENT
4  mov es, ax
5  mov bx, DATA_OFFSET
6  mov word ax, [i]
7  shl ax, PROCESS_DATA_SIZE-1
8  add ax, DISK_DATA_BASE
9  mov cx, PROCESS_DATA_SIZE
10 call DiskWriteSectors
;// over here i is incremented.
..
11 mov ax, PROCESS_SEGMENT
12 mov es, ax
13 mov bx, CODE_OFFSET
14 mov word ax, [i]
15 shl ax, PROCESS_CODE_SIZE-1
16 add ax, DISK_CODE_BASE
17 mov cx, PROCESS_CODE_SIZE
18 call CDRomReadSectors

19 mov ax, PROCESS_SEGMENT
20 mov es, ax
21 mov bx, DATA_OFFSET
22 mov word ax, [i]
23 shl ax, PROCESS_DATA_SIZE-1
24 add ax, DISK_DATA_BASE
25 mov cx, PROCESS_DATA_SIZE
26 call DiskReadSectors
;// over here processor state is loaded
```

Figure 6.8: Total Swapping Implementation

pearance following line 1 is associated with a command that loads a value. The only exception is the variable i, which is verified to be in the range 1-N in lines 1-2. The rest of the code contains three stages for calculating the required parameters for the disk routines. Lines 3-10 save the process state to disk. Immediately after these lines the process number i is incremented and following that, lines 11-18 load the code for the next process from the read-only stable storage, while in lines 19-26 the state of the process is loaded from disk. Afterwards, the processor state for the process is also loaded and the process is finally activated. In more details, lines 3-4 load the register `ex` with a fixed main memory address, pointing to where the running process is residing. Line 5 loads register `bx` with another fixed address which is the offset to the location of the process state.

Lines 6-8 load register `ax` with the disk sector number (address) to write to. First, we copy the value of i to the register. Following that we multiply it by the size of the state block for each process. This size is fixed and assumed here to be a power of 2. Thus the multiplication is carried out by bit shifting. The design is such that the result of multiplying this size by N (the maximum value of i) cannot exceed the register capacity. Finally, the fixed base address on disk where all the states are kept is added. Again contains a value such that the summation will not result in an overflow. Line 9 stores the fixed number of sectors to save to the disk in register `cx`. The actual writing to the disk is performed by the procedure *DiskWriteSectors* (line 10) which is assumed to be atomic in this case. Lines 11-18 and 19-26 load the next process code and state from CD-ROM and disk. The arguments for correctness are exactly like those for lines 3-10.

All addresses in the code are calculated each time. As previously mentioned, except for one variable with checked limits, namely i, the calculations are based on constants. Thus, all the possible addresses are easily verified for containing the correct values. This code is guaranteed to be executed infinitely often. Moreover, the instructions are fixed, thus in the first run (say after a fault), in which the code will be run entirely from the first line, the saving and loading of any process state will be correct. From that time onwards, the processes can stabilize and since they are mutually separated by this full swapping algorithm, the whole system will eventually stabilize too.

6.7.2 Fixed Partition Implementation

The relevant code sections appear in Figure 6.9(a). The code uses procedures for accessing the disk (e.g., `MM_DiskLoadProcessCode`). These procedures calculate arguments for the disk access routings and are omitted here since they are analogous to the ones presented in Section 6.7.1. Lines 1-15 implement the main memory manage-

ment algorithm. Lines 16-29 implement the section of selecting a frame. Swapping is performed in lines 30-41. In line 1, the value of the frame pointed to by the ith entry of PT (which belongs to p_i) is moved to register `ax`. This entry's address is kept in register `bx`. However, this value is repeatedly assigned by the scheduler just before calling the presented code. The frame number is compared first to the NIL value (line 2). If the frame number is NIL, then the process switch can be initiated (line 3). There is also a check for whether the FT entry for this frame number contains i (lines 6-8). This is done first by pointing register `si` to the FT base address (line 4) and then by adding the value of i already kept in `ax` (line 5). According to the conditions above, the procedures for locating a new frame and for swapping the residing process are executed (lines 9-10). In any case, the refresh counter of p_i is decremented (line 11) again based on the correct value of `bx`, which is preserved during the algorithm operation. In case the refresh counter value becomes zero (lines 12-13), the code of the process is reloaded by calling the relevant procedure (line 14). The inputs to this procedure are i, which resides in memory and is periodically checked for pointing to some process, and the selected frame, which was saved in PT and which was validated beforehand (line 7) or was updated by the `Find-Frame` procedure. Line 15 contains the return instruction from the memory manager.

Finding a new frame for the new process involves validating the current frame value of p_i (line 16) and then increasing this value modulo M (lines 17-18) in order to start the search. Yet, these operations are only based on the correct assignment to register `bx` made by the scheduler. Lines 19-20 check if the search ended by moving across the whole of the frame table, while lines 21-25 check in FT whether an empty frame is reached. Note that the check concerning the occupation of a frame by a non-active frame (line 4 of the pseudo-code in Figure 6.4) is omitted here. This does not violate the correctness of the algorithm, but might cause unnecessary swapping of frames. Lines 26-28 increment the new frame counter modulo M and jump back for checking the loop end conditions, as mentioned above. Upon exiting (line 29), the register `al` contains the value of a new frame for the scheduled process.

In order to actually swap out a residing process p_j, we first check that the relevant new frame nf is not marked empty. This is done in lines 30-34 in the same way as before by accessing FT. If nf is not empty, then p_j's state is saved to disk in line 35. The procedure that writes to disk uses the values of nf and j. In case the latter value is corrupted in FT (j is taken from FT), the state of p_j will be corrupted, since the state of the process residing in frame nf will be saved instead of the state of p_j. The process p_j will be scheduled again later, will be loaded with its correct code and will stabilize. For better bookkeeping, the frame field for p_j in PT should be marked NIL (line 3 of pseudo code in Figure 6.4). This is not necessary for correctness, since when rescheduling p_j a contradiction between PT and FT will be found and resolved. Thus, it was

omitted from the current implementation. Lines 36-37 update the nf entry of FT with the new residing process number i (addresses based on the values of i which are maintained by the scheduler and register `si`, calculated in line 30). Line 38 updates the ith entry of PT with the correct frame nf and consequently the state of p_i is finally loaded from disk (line 39). The parameters for this procedure are only i (maintained by the scheduler) and nf, both of which were calculated before. Line 40, implements the decrement of the refresh code countdown, and is also based only on the `bx` value. Line 41 returns from the swap algorithm.

6.7.3 Dynamic Allocation Implementation

Figure 6.9(b) presents the implementation of the garbage collection algorithm of Figure 6.6. Lines 1-3 enforce correct values of segment registers (`ds`, `es`) for correct transfer of data. Lines 4-8 carry the task of zeroing the array in memory which marks the processes that are actively using dynamic memory. This is done by loading the memory address in register `di`, preparing a zero in register `ax` for copying (actually only the lower half is used), and loading the size of the array into register `cx`. Line 7 assures the correct increment of the destination address pointer, by setting the direction flag of the processor. Thus, the whole state is validated towards the actual copy performed in line 8. The `rep` instruction causes the processor's microcode to perform a loop which decrements register `cx` towards zero. In line 9 we assign the variable counter of free segments with zero, It will later be incremented for each segment which will be found free. The final value will be used for deciding whether to assign the free memory to the next waiting process.

Lines 10 to 27 implement the loop which goes over the segment table and decreases the lease for each segment and mark as free a segment the lease of which has reached zero. In line 10 register `cx` is loaded with the size (number of entries) of the segment table. This value will be decremented by the `loop` instruction of line 27 towards zero, and thereafter the operation will move forward. In line 11 the address of the segment table is calculated and stored in register `si`. This register is used in each loop iteration for pointing towards the examined segment. In lines 12-14 we check whether the current segment is owned by some process, In case it is we continue with a lease action. In line 15 the lease for the segment is decreased and is checked in lines 16-17 for a zero value. Note that even if, due to some fault, a process gets an arbitrary lease value. Since the size of the lease entry in memory is bounded, and as long as the segment is occupied, the lease is decremented infinitely often, thus the lease is guaranteed to eventually expire. In case the lease is still valid, lines 18-21 mark the process as using dynamic memory. This is done by taking the base address of this array and adding the value stored in register `al` which was

assigned in line 12 to the holding process number. In case the lease expired, the segment is marked free by assigning a special NIL value at the owner field (line 22). Then, in lines 23-25 we check for such a NIL value (whether achieved by the lease reduction or by a process explicitly releasing a segment), and updating the free segment counter accordingly. Lines 26-27 increment the pointer to the segment table and the loop counter, and as long as it is not zero, jump back to line 12.

In order to complete the garbage collection operation, lines 28-37 traverse the array which is holding for each process the information whether it uses dynamic memory. Lines 28-29 load register `si` with the address of the array. They also load register `di` which serves as an index to zero. Line 30 stores the size of the array in register `cx`. This value will be decremented towards zero by line 37. In lines 31-32 we check for zero value in the array for each process. In case of a zero value, we mark in the process table entry for each process the fact that dynamic memory is not being used. Thereafter, this process will not be able to use this segment, unless allocated again. Line 33 loads the base address of the process table in memory and line 34 updates the relevant field by adding an index and an offset to the process' row. By advancing the indices, the next iteration is prepared in lines 35-36.

The procedure ends with an attempt to allocate the available dynamic memory to a waiting process. Line 38 points register `si` to the memory holding the queue. This area contains requests for memory allocation containing the requesting process id, the requested quantity and a lease period. In lines 39-41 we check whether the queue is empty and if it is, there is no more work to be done. Lines 42-45 check whether the top request size is larger than the available segments. If that is the case there will be no further action (until additional space is freed). Note that line 43 validates that the request size is not larger than the whole dynamic memory size. Otherwise, an error in that value might prevent all future allocations (this argument does not hold for the lease parameter which is designed to take care of any possible value. This value is bounded only by the size of the containing variable). Finally in lines 46-48 we are in a state in which a new allocation can be made. The lease expiration value is prepared in register `ch` and we call the two utility procedures which pop the request from the queue and assign the necessary free memory.

Figure 6.10(a) presents the implementation for the `MM-Alloc` procedure, called by the processes in order to obtain dynamic memory, and also the internal `_MM-Assign` procedure. (The other routines e.g., `MM-Free`, that are mentioned in the pseudo code, are simpler to implement and are mainly used for performance optimizations, therefore were omitted from the current implementation version). Figure 6.10(b) completes the implementation with the queue handling procedures. The code corresponds directly to the pseudo code algorithms presented in Section 6.5. Lines 1-3 of `MM-Alloc` insure that the requesting process does not hold any dynamic memory already any more. This

is ensured by checking the **fs** register which only points to a dynamic segment address for granted processes. Lines 4-6 prepare the required parameters for calling the `_MM- Assign` procedure (line 9) and check if substantial memory is available. Otherwise, the enqueue procedure is called for in line 7. The assignment procedure sets up a loop (lines 11-13) for traversing the memory segment list. Then, each list entry is checked for emptiness (lines 14-19). In case an empty entry is found, it is marked as owned by the requesting process and the lease is recorded too (lines 20-21). Line 22 decrements the global counter for free segments. Then, in lines 23-25 a check is carried out in order to see whether enough allocations had already been made. Otherwise the loop continues through lines 26-27. Lines 28-30 update the segment register pointer with the new allocation. Since this procedure can also be called independently by the validator process (lines 31-36), this update is carried to the process' state in the process table too.

The `_MM-Enqueue` operation of Figure 6.10(b) works as follows. In lines 1-10 the queue is searched for the requesting process. If found, the operation stops in line 7. In lines 11-23 an empty slot is searched for placing the request. The `_MM-Deque` operation is carried out by traversing the queue (lines 25-37) and by advancing each slot by one location. In order to mark it as free, line 38 places a `NIL` value at the end of the queue.

```
MM_SelectNextProcessAndFrame:
1  movzx ax, byte [bx+FRAME_COL]
2  cmp al, NULL_FRAME
3  jz StartProcessSwitch
4  lea si, [FT]
5  add si, ax
6  mov al, byte [si]
7  cmp al, byte [i]
8  jz BypassProcessSwitch
StartProcessSwitch:
9  call MM_FindFrame
10 call MM_SwapProcess
BypassProcessSwitch:
11 dec byte [bx+RELOAD_COL]
12 cmp byte [bx+RELOAD_COL], 0x0
13 jnz BypassLoadProcessCode
14 call MM_DiskLoadProcessCode
BypassLoadProcessCode:
15 ret
MM_FindFrame:
16 and byte [bx+FRAME_COL], FRAME_MASK
17 inc al
18 and al, FRAME_MASK
while1:
19 cmp al, [bx+FRAME_COL]
20 jz endwhile1
21 lea si, [FT]
22 add si, ax
23 mov dl, [si]
24 cmp dl, NULL_TASK
25 jz endwhile1
;missing: and active[PT[FT[nf]]] = true
26 inc al
27 and al, FRAME_MASK
28 jmp while1
endwhile1:
29 ret
MM_SwapProcess:
30 lea si, [FT]
31 add si, ax
32 mov dl, [si]
33 cmp dl, NULL_TASK
34 jz NoDiskSave
35 call MM_DiskSaveProcessData
;missing: frame[PT[FT[nf]]] := nil
NoDiskSave:
36 mov cx, word [i]
37 mov byte [si], cl
38 mov byte [bx+FRAME_COL], al
39 call MM_DiskLoadProcessData
40 mov byte [bx+RELOAD_COL], 1
41 ret
```

(a) Fixed Partition Implementation

```
1  mov ax, DATA_SEGMENT
2  mov ds, ax
3  mov es, ax
4  lea di, [usingDynamicArray]
5  mov ax, 0
6  mov cx, NUM_PROCESSES
7  cld
8  rep stosb
9  mov byte [freeSegments], 0
10 mov cx, NUM_SEGMENTS
11 lea si, [SegmentTable]
FOR1:
12 mov al, byte [si+OWNER_COL]
13 cmp al, NIL_PROCESS
14 je ENDIF1
15 dec byte [si+LEASE_COL]
16 cmp byte [si+LEASE_COL], 0
17 je IF2
18 lea bx, [usingDynamicArray]
19 add bx, ax
20 mov byte [bx], 1
21 jmp ENDIF2
IF2:
22 mov byte [si+OWNER_COL],NIL_PROCESS
ENDIF1: ENDIF2:
23 cmp byte [si+OWNER_COL],NIL_PROCESS
24 jne ENDIF3
25 inc byte [freeSegments]
ENDIF3:
26 add si, SEGMENT_TABLE_ENRTY_SIZE
27 loop FOR1
28 lea si, [usingDynamicArray]
29 mov di, 0
30 mov cx, NUM_PROCESSES
FOR2:
31 cmp byte [si], 0
32 jne ENDIF4
33 lea bx, [processTable]
34 mov word [bx + di + SEG_COL], NIL_SEG
ENDIF4:
35 inc si
36 add di, PROCESS_ENTRY_SIZE
37 loop FOR2
38 lea si, [queue]
39 mov al, byte [si + PROCESS_COL]
40 cmp al, NIL_PROCESS
41 je ENDIF5
42 mov cl, byte [si + QUANTITY_COL]
43 and cl, NUM_SEGMENTS_MASK
44 cmp cl, [freeSegments]
45 ja ENDIF5
46 mov ch, byte [si + EXPIRATION_COL]
47 call _MM_Deuque
48 call _MM_Assign
ENDIF5:
```

(b) Dynamic Memory Validator Implementation

Figure 6.9: Fixed and Dynamic Validator Implementations

```
MM_Alloc:
1 mov dx, fs
2 cmp dx, 0
3 jne ENDIF1
4 mov al, byte [processIndex]
5 cmp cl, [freeSegments]
6 jbe ASSIGN
7 call _MM_Enqueue
8 jmp ENDIF1
ASSIGN:
9 call _MM_Assign
ENDIF1:
10 ret

_MM_Assign:
11 mov dx, 0
FOR1:
12 cmp dx, NUM_SEGMENTS
13 jae ENDFOR1
14 lea si, [SegmentTable]
15 mov bx, dx
16 shl bx, SEGMENT_TABLE_ENRTY_SIZE_EXP
17 add si, bx
18 cmp byte [si+OWNER_COL], NIL_PROCESS
19 jne ENDIF2
20 mov byte [si+OWNER_COL], al
21 mov byte [si+LEASE_COL], ch
22 dec byte [freeSegments]
23 dec cl
24 cmp cl, 0
25 je ENDFOR1
ENDIF2:
26 inc dx
27 jmp FOR1
ENDFOR1:
28 shl dx, SEGMENT_WIDTH
29 add dx, SEGMENT_BASE
30 mov fs, dx
31 lea bx, [processTable]
32 movzx dx, al
33 shl dx, PROCESS_ENTRY_SIZE_EXP
34 add bx, dx
35 mov dx, fs
36 mov word [bx + SEG_COL], dx
37 ret
```

(a) Dynamic Memory API Implementation

```
_MM_Enqueue:
1 mov dx, 0
2 lea si, [queue]
FOR5:
3 cmp dx, NUM_PROCESSES
4 jae ENDFOR5
5 cmp byte [si+PROCESS_COL], al
6 jne ENDIF8
7 ret
ENDIF8:
8 add si, QUEUE_ENTRY_SIZE
9 inc dx
10 jmp FOR5
ENDFOR5:
11 mov dx, 0
12 lea si, [queue]
FOR6:
13 cmp dx, NUM_PROCESSES
14 jae ENDFOR6
15 cmp byte [si], NIL_PROCESS
16 jne ENDIF9
17 mov byte [si + PROCESS_COL], al
18 mov byte [si + QUANTITY_COL], cl
19 mov byte [si + EXPIRATION_COL], ch
20 jmp ENDFOR6
ENDIF9:
21 add si, QUEUE_ENTRY_SIZE
22 inc dx
23 jmp FOR6
ENDFOR6:
24 ret

_MM_Deuque:
25 lea si, [queue]
26 mov dx, 0
FOR7:
27 cmp dx, NUM_PROCESSES-2
28 ja ENDFOR7
29 mov ah, byte [si+
   QUEUE_ENTRY_SIZE+PROCESS_COL]
30 mov byte [si+PROCESS_COL], ah
31 mov ah, byte [si+
   QUEUE_ENTRY_SIZE+QUANTITY_COL]
32 mov byte [si+QUANTITY_COL], ah
33 mov ah, byte [si+
   QUEUE_ENTRY_SIZE+EXPIRATION_COL]
34 mov byte [si+EXPIRATION_COL], ah
35 add si, QUEUE_ENTRY_SIZE
36 inc dx
37 jmp FOR7
ENDFOR7:
38 mov byte [si+QUEUE_ENTRY_SIZE],
     NIL_PROCESS
39 ret
```

(b) Dynamic Memory Helper Procedures

Figure 6.10: Dynamic Memory Implementation

Chapter 7

I/O Device Drivers

7.1 Introduction

Device drivers are known to be a major cause of operating system failures [30, 136] for a variety of reasons. First, drivers are usually loaded into the operating system kernel's address space and are running in privileged processor modes where an error has a greater effect on the total system behavior. Additionally, usually essential system parts are designed, built, verified, and tested with extra care while many times drivers are brought from the outside. The following techniques are used to deal with these failures: (a) reducing the driver's access to system resources [73, 139], (b) containment of errors in realtime through kinds of virtualization [13, 136], (c) using typed languages [73, 126], and (d) static analysis of the drivers' code, and of their resource use [12, 133]. Applying such techniques helps improve the system's robustness, but the bottom line is that in systems running for a long period of time, errors (e.g., soft errors [105]) in device drivers accumulate and lead to undesired behavior.

Device drivers (or simply drivers) are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing a consistent and more abstract interface between other programs and the hardware device resources (sometimes they also add extra services not provided by the hardware devices). I/O devices usually contain a controller which is the electronic part with which drivers communicate. The communication is carried out via the system bus, and is usually done through some standard protocols and interfaces e.g., ATA and SCSI for disk drives. In [114] it is stated that "a modern Seagate drive contains roughly 400,000 lines of code." In [139] it is noted that "modern disk controllers often have many megabytes of memory inside the controller." The complexity of today's I/O devices emphasizes the need for robust device

drivers.

Here, we suggest enhancing the robustness of device drivers by designing them to be self-stabilizing. Building a system, and specifically device drivers, in a self-stabilizing way, ensures that errors will be contained autonomously by each driver, leading eventually to correct behavior of the whole system.

Implementation issues. A demonstration implementation using the Intel Pentium processor architecture [75] was composed. This implementation is written in assembly language and is directly assembled into the processor's opcode. The reader may choose to skip the implementation details. Our proofs and prototype show that it is possible to design and implement self-stabilizing device drivers that preserve the stabilization of the running programs — an important building block of an infrastructure for industrial self-stabilizing systems.

Related work. As explained earlier, robustness of device drivers is of great importance in system design. It is stated in [30] and [90] that about 70% of operating system code is devoted to device drivers. It is stated in [136] that "In Windows XP, for example, device drivers cause 85% of reported failures." Moreover, in [30] it is claimed that for some cases in Linux, "the error rate for drivers is almost seven times higher than the error rate for the rest of the kernel." Here we briefly survey various efforts in this field.

• **Device driver isolation and monitoring.** The micro-kernel system architecture (pioneered in the Mach system [2]) suggests achieving a minimal trusted computing base (TCB) by removing as much as possible from the kernel. For example, in the last version (3) of Minix [139], the drivers' access to system resources is restricted. This was achieved by factoring the common low level and privileged commands, such as access to I/O ports and interrupts, and moving most driver parts to user space, where they communicate with the kernel through a simple message mechanism (see [142] for another version).

• **Virtualization.** A variation of this approach, lately suggested by many, is to run the original drivers of common operating systems, but to monitor their activity and contain errors by different kinds of virtualization [13, 89, 90, 136]. This method counts heavily on the robustness of the core kernel (also known as Virtual Machine Monitor), which we actually address in this work. In [88, 94] and [89] this is combined with an IO-MMU which adds hardware protection to I/O access. In [13] it is claimed that this method is not enough, and "in the case of more 'sophisticated' statefull devices it may be in addition necessary to reset the device to a known state." In [135], a monitor which

records the inputs sent to a driver by an application is added. In case of a failure a restart of the driver is carried out together with replaying the inputs.

• **Type safety and model checking.** In Coyotos [126], the whole kernel, including drivers, is written in a typed language as a stage towards achieving formal correctness. Static analysis of the drivers' code appears in [11, 12]. Recently those methods were augmented with a general tool for termination checking [31] that is used mainly to check device drivers. In [12] it is claimed that kernel APIs are usually too complex so there is a great chance for coding bugs. They categorize bugs in order to find them automatically. This emphasizes the need for a good understanding of the protocols between drivers and the rest of the system. Many others (e.g., [30]) use code analysis to find kernel bugs in general.

• **Singularity** is a recent ongoing research project [72, 73] which combines many of the past system research advances in order to achieve greater system dependability. In Singularity, drivers are also treated as user programs so their state is separated from the rest of the system. Hardware resources are accessed only through messages, and when the system is compiled or started, there is a verification process carried out according to meta-data resource declarations. In [133], details concerning device drivers are provided. This project also relies on a typed language to restrict drivers abilities. To prevent malicious code behavior, runtime code changes are restricted by eliminating language features such as reflection. On the other hand, all programs in Singularity run in privileged mode. These settings do not prevent a transient error from corrupting system execution (to quote [73], a "malicious driver can program a DMA capable device to overwrite any part of memory.")

None of the above suggest device driver design and implementation that can automatically recover from an arbitrary state (that may be reached due to a combination of unexpected faults and sequence of unexpected inputs).

Chapter Organization. In this chapter we demonstrate how device drivers can be designed to be self-stabilizing. We start, in Section 7.2, by demonstrating how the current ATA specification for storage devices (such as hard disks) requires a behavior which can lead a system into undesirable combined states. Based on the definitions and settings presented in Section 7.3, we demonstrate in Section 7.4 how the design can be augmented to behave in a self-stabilizing way. Proofs for the correctness of the suggested solutions are provided. Concluding remarks are given in Section 7.5. An explained implementation which fulfills the self-stabilization requirement appears in the appendix, Section 7.6.

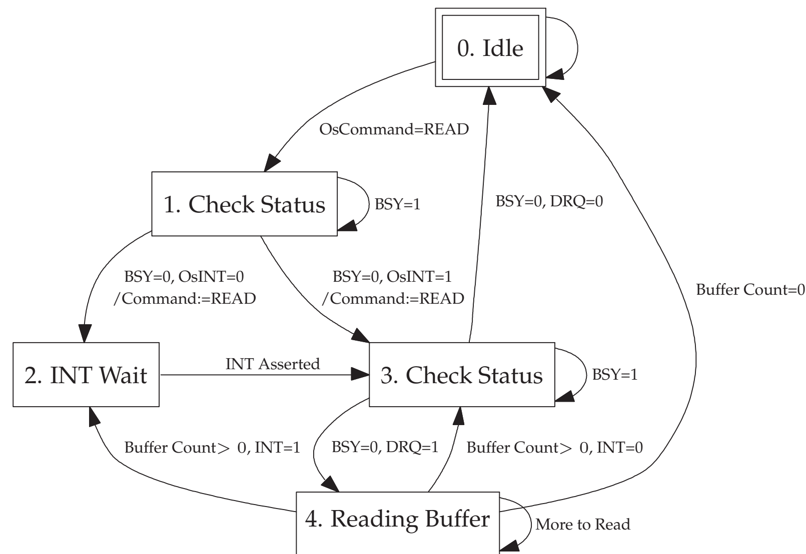


Figure 7.1: ATA Host State Transitions.

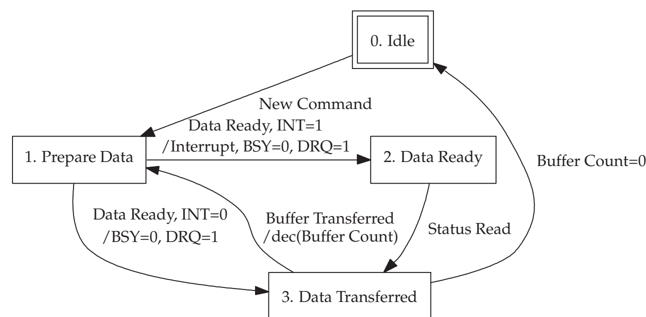


Figure 7.2: ATA Device State Transitions.

7.2 A Non-Self-Stabilizing Driver Specification

The AT-Attachment protocol (standard draft version 8, also historically known as IDE) defines a parallel transport protocol between *host* systems and *device*s [137]. In the following we will first describe this protocol. Then we will show that the protocol defines interactions which can lead to non-stabilizing executions. Note that the standard defines only the *interface* between a host and a device. Therefore, an implementation can add states and transitions in order to achieve stability, and still conform to this standard.

Communication between the host and the device is by means of input/output registers (the shared memory model). There are control, command, status, and data registers through which the host and the device communicate. Addition-

ally, the device might signal the host through an interrupt line. The required behavior is defined with state diagrams describing states and transitions of both protocol parties. For the purpose of demonstrating the non-stability, we follow diagrams describing the execution of a read command. In order to carry out such a command, the two parties move from an idle state to the executing command states and upon completion back to the idle state. The idle states of the host and the device, respectively are described in Figures 41 and 43 in the current specification [137]. The PIO (Programmed I/O) data-in command states, which transfer blocks of data from the device to the host, without using DMA (Direct Memory Access) are described in Figures 47 and 48. We combine these four diagrams into two state machine diagrams each describing the possible executions of the host and the device, respectively. In general, upon a read request, the host checks whether the device is ready (state 1 of Figure 7.1); it then configures the device, writes the command parameters, and waits for response through an interrupt (state 2) or by repeatedly checking status (state 3). The device fills its transfer buffer with part of the requested data (state 1 of Figure 7.2) and signals the host for availability (states 2 and 3) by asserting the interrupt line or setting the data request (DRQ) status bit. The host then reads this data (host state 4) and the interaction continues until completion (buffer count reaches zero), when they both return to their idle state (state 0). For this demonstration we omit many technical details, e.g., selection of devices and media error handling. We show that even if we make the model simpler and also assume perfect operation of the device mechanics, the execution can still become erroneous.

7.2.1 Non-stability

The model above does not assume a behavior in which progress is achieved infinitely often. Even assuming correct behavior of each party, the combined execution can enter states in which progress is not achieved infinitely often. The various combinations of states that the system can reach are presented in Figure 7.3. The arrowed path demonstrates a possible correct execution which is cyclic and includes the combined idle state (marked “h0d0”). Possible non-stabilizing executions are described next: **Deadlock**. Due to a transient error, the combined state of the system can change to a state which is not part of the path described above — a state from which there is no defined progress. Such a scenario, for example, is one in which execution reaches the combined host state 2 with device state 3. In host state 2, the host waits for an interrupt in order to transfer data. In the meantime, the device (say, due to a transient error) assumes that interrupts are disabled and waits in its state 3 for the host to read data from its buffer. From such a combination of states there is no defined progress.

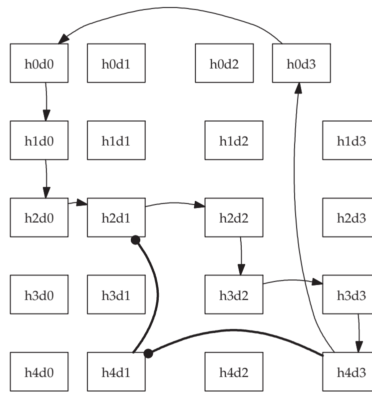


Figure 7.3: Union State Transitions (h=host, d=device).

Livelock. Another case is demonstrated in Figure 7.3 where the dot-headed path causes the execution to circle back to state “h2d1” without ever bringing to an end the current command execution. This happens when the host is cycling between states 2 and 4, reading the device’s buffer content, but the buffer counter never goes down to zero.

Another example is when the host waits falsely for the device (e.g., in state “h1d3”). It reads in the state register that it is busy, while the device is really non-busy and actually ready to proceed.

Corollary 7.2.1. *The* ATA *protocol is not self-stabilizing.*

The standard also addresses some other scenarios. For example, if a command is issued by the host while the device is busy with a previous command, the device should immediately start executing the new command. However, we require a design in which the system converges from any combined state.

7.3 System Model and Requirements

Settings. We divide the system into four parts: (a) The *operating system*, which contains *processes* (or programs) which can request I/O operations. The operating system contains a special program which schedules all the various processes, including part *b*. (b) The *operating system device driver* (or OS *driver*) is the special program which handles the I/O requests and communicates with the device. (c) The device *controller* is the program executed by a specialized micro-processor (it usually resides inside the I/O device itself) which commands the I/O device to perform its task. (d) The (I/O) *device* is the actual peripheral machinery that carries out the commands, e.g., rotating the disk media under one of its reading heads.

(a) and (b) together map to the *host* in the ATA specification while (c) and (d) are mapped to the *device*.

Assumptions. We concentrate on the correct behavior and interactions of one OS driver (b) and the corresponding device controller (c). Thus the state diagrams presented in Figures 7.1 and 7.2 are considered transitions made by the OS driver and the device controller, respectively. Concerning the operating system (a), using methods described in Chapters 5 and 6, we assume that the operating system is self-stabilizing. It is particularly guaranteed that, during the system execution, whenever there are pending I/O requests, the scheduler will eventually execute the driver program, thus allowing it to operate as required. Fair access between processes with regard to the ability to queue I/O requests is achieved either by assuming eventual correct behavior of the processes (we do not assume the Byzantine model) or by leasing the right to queue messages in ways that will guarantee fairness as done by the memory manager of Chapter 6.

It is also assumed that the I/O device's micro-processor is self-stabilizing, which means that it keeps fetching and executing the device controller program. Methods to achieve such behavior are described in [43]. Additionally, the device mechanics (or other equivalences in other devices) always eventually respond to the device controller commands either by carrying them out or by reporting an error in case of, say, physical disabilities, e.g., bad sectors on the disk media.

The OS driver and the device controller communicate by writing in each others registers. It is assumed that every read/write operation is performed atomically and without errors.

Definitions. We describe briefly a set of definitions related to states and state transitions (again, see Chapter 3 for details concerning processor executions, interrupt and register settings, and additional requirements). A *state* of the operating system driver or the device controller is an assignment to its registers including the program counter resister. Each party is modeled by a program which specifies its behavior. It has a clock which triggers a *step* which is a state transition. The transition is done according to the current state (including input registers and the program counter). A *configuration* is a pair of states, the first of which is of the OS driver and the second to the device controller. An example of such a configuration is "h0d0" which appears in Figure 7.3 in which both parties are in their idle state. An *execution* is a sequence of alternating configurations and steps $E = (c_1, s_1, c_2, s_2, ...)$, such that configuration c_{i+1} is reached from configuration c_i by one step s_i taken by one of the parties. A configuration like "h0d0" is called a *safe configuration* since an execution that starts from this configuration carries out the task of executing I/O commands correctly.

The various register roles used in the described read command are listed for each protocol party in Table 7.1.

Owner	Register	I/O	Role
OS driver	OsCommand	I	Command parameters written by OS
	OsINT	I	Interrupt configuration written by OS
	INT Line	I	Interrupt line* asserted by device controller (*Not a register)
Device controller	Command	I	Command parameters written by OS driver
	INT	I	Interrupt status written by OS driver
	BSY	O	Controller working status read by OS driver
	DRQ	O	Data ready status read by OS driver
	Buffer Count	I/O	Data read status written by OS driver and decremented by device controller

Table 7.1: ATA Registers.

The Error Model. The OS driver and the device controller states, including their program counters, might become corrupted (assigned any possible value).

Requirements. We now define the requirements which should be satisfied for the described system to be self-stabilizing.
(r1) Ping-pong. Assuming that there is an infinite system execution, in which there are infinitely many I/O requests, the OS driver and the device controller are infinitely often exchanging requests and replies.
(r2) Progress. Eventually every I/O request is executed completely and correctly according to the ATA specification. As explained, the result can be a success, e.g., data moved according to the command's parameters, or a failure due to bad parameters, some transient error (such as dust on the disk surface), or even non-transient device errors such as bad sectors.

A *self-stabilizing* OS *driver and device controller combination* ensures that every infinite execution of a system has a suffix in which both requirements hold.

7.4 A Self-Stabilizing Driver

The OS driver and the device controller can be viewed as a master and a slave working together according to a protocol to achieve their mission. Thus, the driver acting as a master can check that the slave is following, say the ATA protocol, correctly. We suggest two solutions. In the first solution the device controller is not required to be self-stabilizing, and the OS driver leases the device controller some (usually enough) time to complete its tasks. Then we relax the timing constraints by assuming that the device controller itself is also self-stabilizing. Therefore we only need to guarantee that the execution is

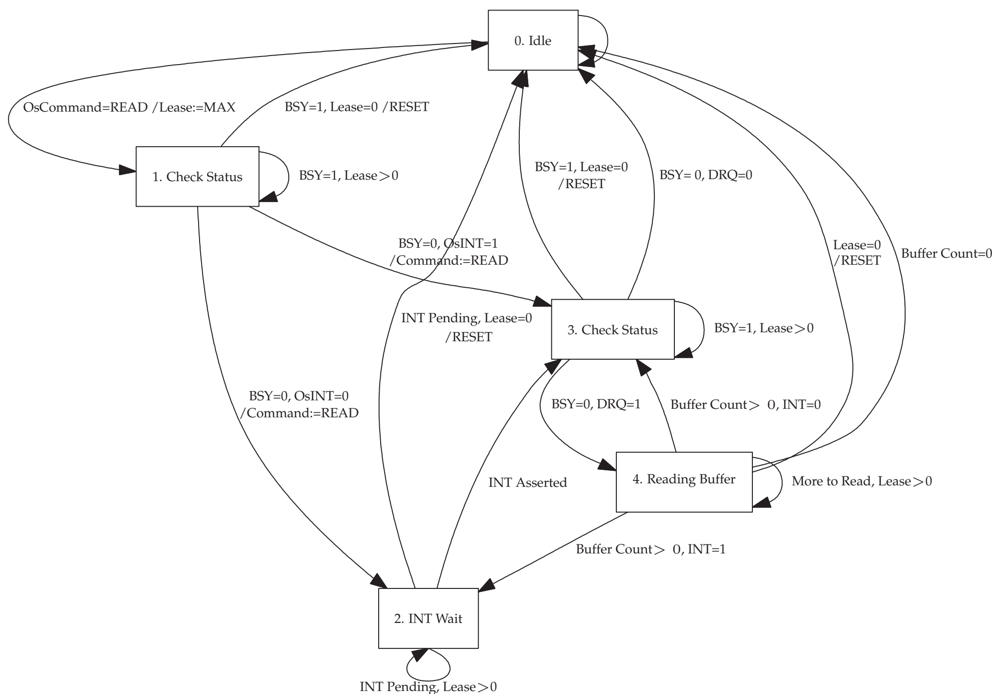


Figure 7.4: A Leasing Host (OS Driver).

carried out by both parties according to the protocol. This is achieved by the OS driver performing consistency checks according to its current state. Note that the device controller itself is working against the underlying device so an implementation of the driver-controller protocol can use either the leasing or the consistency check solutions for this level as well.

7.4.1 Leasing

In order to satisfy our requirements we suggest that the device controller should be augmented by a counter register which is used to implement a watchdog. The OS driver is able to write some value to this register, while the device hardware is lowering the register value towards zero, say in every clock tick. Additionally, the OS driver is augmented with additional transitions which guarantee that in case the lease expires, which means that there is no progress from the device controller side, the OS driver resets the device controller's state and also moves to the idle state. The new OS driver transitions are described in Figure 7.4.

Next we prove the correctness of the leasing solution.

Lemma 7.4.1. *The* OS *driver reaches its idle state infinitely often.*

Proof. The OS driver always converges towards the idle state. This can be observed by examining the possible state transitions. We can see that in every state the OS driver can either move to a higher numbered state (modulo the number of states) or stay in the current state. The only exception to this rule is the move from state 4 back to states 2 or 3 (depending on the interrupt status), but the number of such possible backward moves is bounded by the countdown of buffer reads, which is decreased towards zero every time such a back move is taken. So the OS driver is guaranteed either to proceed through the protocol stages and eventually reach the idle state, or else to get stuck in some state. In the latter case, since the OS driver stopped leaving the idle state, the only state where it updates the device leasing counter, this register will eventually reach zero causing the OS driver to perform a move to the idle state. □

Lemma 7.4.2. *From any configuration a safe configuration is eventually reached.*

Proof. From Lemma 7.4.1, the OS driver reaches the idle state infinitely often. Usually according to the protocol design, the device controller will reach the idle state following the OS driver. Whenever the OS driver starts a new command it advances along the protocol stages waiting for the device controller to follow it as they carry out the I/O request together. Whenever a command is completed both parties proceed to the idle state, which is a safe configuration. Otherwise, the OS driver will eventually reset the device controller to its idle state, and also will move to its own idle state, thus again reaching a safe configuration ("h0d0"). Note that in case of successive I/O commands the OS driver might wait in state 1 for the controller to finish the last command and join it. Therefore, the configurations "h1d3" and "h1d0" are safe configurations as well. □

Corollary 7.4.3. *Since a safe configuration is reached from any state, ping-pong holds.*

Lemma 7.4.4. *Eventually progress holds.*

Proof. From Lemma 7.4.2, a safe configuration is eventually reached. From this configuration, the OS driver and the device controller fulfill requests according to the protocol specification. □

Corollary 7.4.5. *Since in every infinite system execution the ping-pong and progress requirements hold infinitely often, the* OS *driver and device controller combination is self-stabilizing.*

7.4.2 Consistency Checking

Alternatively, if we can assume stabilization of the device controller, then it suffices to guaranty that the device follows the OS driver while executing commands. The OS driver will be augmented with a consistency checker routine

that checks the consistency of both parties. The timing of the execution of this routine can be tuned to occur before each driver code execution, or periodically by means of a watchdog timer and a non-maskable interrupt as described in Chapter 3. The routine freezes the OS driver and reads its program counter register. It also interrupts the device controller, which stops all activity and then reads a snapshot of the device controller's state. The routine then ensures that the controller is in a proper state according to the driver stage of the protocol. In case of consistency violation, actions are taken e.g., resetting the controller.

For each (abstract) state of the diagrams presented in Figures 7.1 and 7.2, there can actually be a set of program counter values which fits that state. The programs of the OS driver and the device controller can be assembled in such a way that there is a simple function which maps every program counter value to its corresponding diagram state. We say that every diagram state is represented by a Program Counter Segment (PCS). This can be implemented, for example in the Intel IA32 architecture [75], by allocating a full code segment for each PCS (more details appeared in Chapter 3).

OS Driver PCS	Device Controller Snapshots
0,1	PCS=0 PCS=3
2	PCS=1, INT=1 PCS=2, INT=1
3	PCS=1 PCS=2, INT=1 PCS=3, INT=0
4	PCS=3

Table 7.2: Consistency Check Rules.

The legal device controller PCS and other register values for every such OS driver PCS are included in Table 7.2. For example, when the OS driver is waiting for an interrupt (state 2), the device controller must be in states 1 or 2 but not in state 3, where it could wait for the driver forever. The interrupt status of the device controller must also be checked to ensure that the device controller will inform the OS driver upon completion. As to other registers, such as DRQ, there is no need to check consistency since the self-stabilization of the device controller guarantees that it will eventually (and in bounded time) set the required values needed for the execution to progress according to the protocol.

Lemma 7.4.6. *Eventually ping-pong holds.*

Proof. Similarly to Lemma 7.4.1, the OS driver advances along the protocol stages. In every state it can either advance to the next stage or wait for the

device controller. Since we have the consistency checker assuring that the device controller is in a proper matching state, and since the device controller is self-stabilizing, eventually the device controller will perform the current stage and the OS driver will advance to the next stage. □

Lemma 7.4.7. *Eventually progress holds.*

Proof. Similar to the proof of Lemma 7.4.4. □

Corollary 7.4.8. *Since in every infinite system execution the ping-pong and progress requirements hold infinitely often, the* OS *driver and the device controller combination is self-stabilizing.*

7.5 Concluding Remarks

Self-stabilization methods enhance the robustness of device drivers, and consequently of their included systems. We demonstrated the lack of such properties in one of the well-known standard protocols. The two solutions that were proposed can be practically combined according to the level of stabilization that can be expected from various I/O devices. If not all the device producers can be relied upon, then one can rely on leases and restarts to achieve self-stabilization. In the other case, the snapshot of the I/O device, taken during a consistency check, can be reduced if its stabilization ability is enhanced. A demonstrating implementation of a hard-disk driver is presented. We have tested it using the BOCHS [20] simulator, and observed that even when completely changing the contents of the RAM, stabilization is achieved. The full implementation can be found in [132].

7.6 Appendix. Hard-Disk Driver Implementation

An example of implementation of a hard-disk driver is presented here. For better readability the code is divided into Figures 7.5(a) through 7.13. Each code portion is briefly explained and arguments for its stability are given. This driver implements the lease-based solution of Section 7.4.1. We mark with the notion "`ATA X=>Y`" the places where the code implements a transition from state `X` to state `Y` according to the augmented ATA host state diagram which appeared in Figure 7.4. Note that some "C" language line comments in this source, which start with the pattern ";//", are added for extra explanation. This is from the source of the ROM-BIOS of Mandrake-Linux which was adopted to PC emulators such as Bochs [20].

The hard-disk functionality is accessible to the other programs through calling API functions such as `HDD_ReadSectors` and supplying arguments through the processor's registers. Those functions act as entry points for the driver functionality.

Each such function converts the logical addresses given in its arguments to disk sector and track addressing and then calls the disk services through soft interrupts (`ATA 0=>1`). In case of an error, such as a busy disk, it tries again several times, up to some predefined value (`ATA 1=>1`) and if it does not succeed, it reboots the device, thus bringing it to the "idle" state from where it is assumed to behave correctly (`ATA 1=>0`). We now go into details of the presented code that forms the stabilizing device driver.

7.6.1 Verifying Stack Memory Coherence

In lines 1-17 of Figure 7.5(a) we ensure that the stack memory area is within its correct limits and then move on to the rest of the code. This is necessary since the stack is used heavily for storing parameters and intermediate results. The rest of the code uses this memory within the checked limits only, thus we guarantee that the code will be executed fully without memory access exceptions. In cases where a transient error, say in the stack pointer value, causes such an invalid access, the exception mechanism of the system is designed to fully recover into a legal state. Whenever the execution of the issued command is finished (successfully or stopped due to an error), the state of the stack is preserved at its original value. Then all register values are preserved and the `iret` instruction transfers control back to the calling program.

7.6.2 Parameter Calculations and Validations

The code in Figure 7.5(b) calculates the base memory address of the Extended Bios Data Area (EBDA) which holds hard disk parameters (these parameters are usually copied from CMOS during startup, but here we assume

```
1  test sp, STACK_LIMIT
2  jae continue0
3  mov sp, STACK_LIMIT
continue0:
4  mov word [ss:STACK_LIMIT], ax
5  mov ax, STACK_SEGMENT
6  mov ss, ax
7  mov ax, word [ss:STACK_LIMIT]
; pass all parameters on stack
8  pusha
9  push es
10 push ds
11 push ss
12 pop ds
13 call int13_harddisk
14 pop ds
15 pop es
16 popa
17 iret
```

(a) Verifying Stack Memory.

```
int13_harddisk:
19 push bp
20 mov bp, sp
21 add sp, 0xfffa ; sp-=6
; read ebda segment
22 mov ax, 0xe
23 push ax
24 mov ax, 0x40
25 push ax
26 call read_word
27 add sp, 0x4
; save result on stack
28 mov word [ss:bp+0xfffa], ax
29 add sp, 0xffe4
;// clear completion flag
;// basic check : device has to be defined
;// basic check : device has to be valid
```

(b) Getting base BIOS address and skipping basic checks.

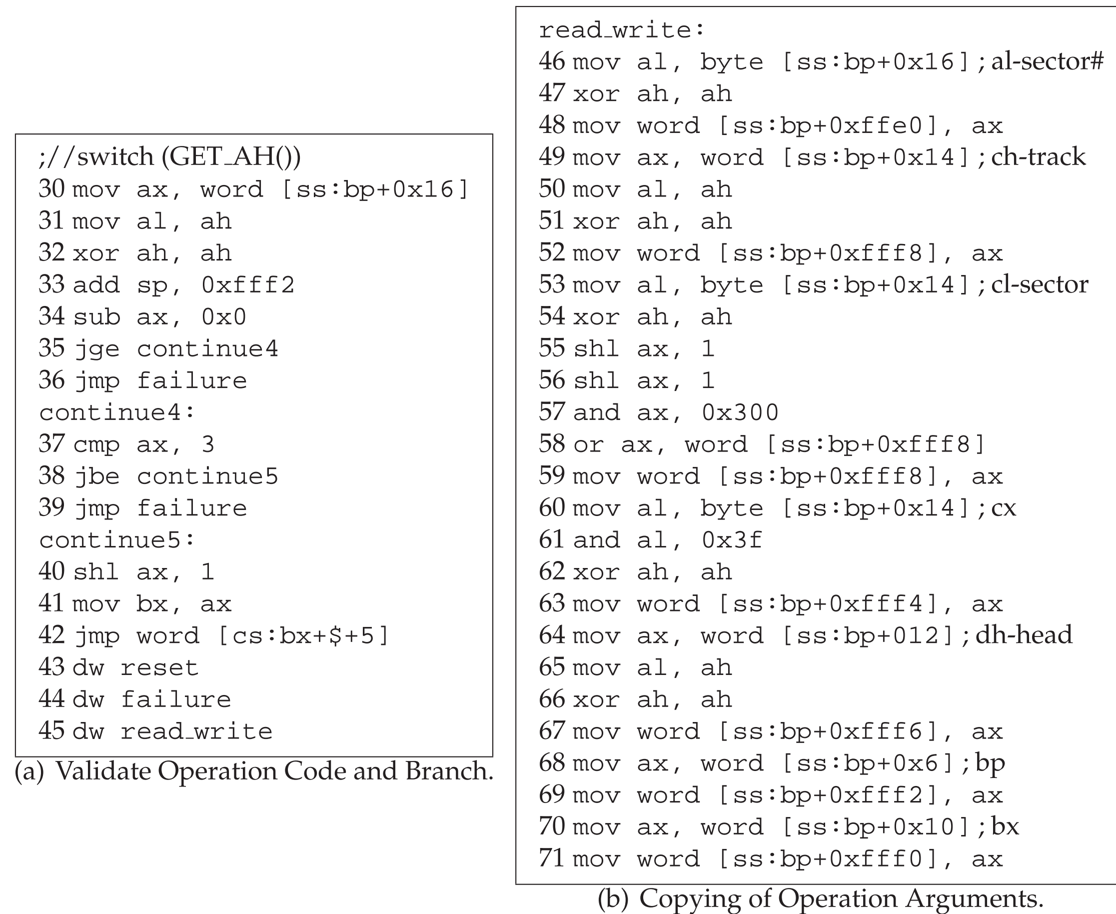
Figure 7.5: HD Driver Impl. 1-2

they are fixed). In lines 19-29 we use a recurring pattern for reading (or writing) a value elsewhere in memory, by pushing the target address to the stack and calling a routine in a set of family of routines for reading\writing bytes and words from\to memory (line 26). The last remark lines show that in this implementation we can skip some checks, e.g., checking that the drive is present and valid, since it is assumed to be enforced by external means.

The code in lines 30-45 (Figure 7.6(a)) validates that the disk command code is in the range of available commands and jumps to the fixed address of the appropriate command (e.g., reset or read/write). Here we are not supporting the dozens of commands usually available, so instead of the computed goto we could check for each possible command code and branch to the appropriate fixed address. Even in the unlikely event in which the operation value is being corrupted right after the check is made and before the branch in line 42 (thus causing the processor to jump to unpredicted address) still the NMI architecture (see Chapter 3), which the system is based on, will eventually recover the system state to a correct state.

In Figure 7.6(b) the various operation parameters are just copied inside the stack within the limits mentioned above.

The code in Figure 7.7(a) (lines 72-78) verifies that the number of requested sectors to read/write is legal, i.e., between 0 to 127. If this is not the case, there is a branch to a failure routine (appears in Figure 7.13) which currently resets the system. The "C" code returns an error value which is much less



```
;//switch (GET_AH())
30 mov ax, word [ss:bp+0x16]
31 mov al, ah
32 xor ah, ah
33 add sp, 0xfff2
34 sub ax, 0x0
35 jge continue4
36 jmp failure
continue4:
37 cmp ax, 3
38 jbe continue5
39 jmp failure
continue5:
40 shl ax, 1
41 mov bx, ax
42 jmp word [cs:bx+$+5]
43 dw reset
44 dw failure
45 dw read_write
```

(a) Validate Operation Code and Branch.

```
read_write:
46 mov al, byte [ss:bp+0x16];al-sector#
47 xor ah, ah
48 mov word [ss:bp+0xffe0], ax
49 mov ax, word [ss:bp+0x14];ch-track
50 mov al, ah
51 xor ah, ah
52 mov word [ss:bp+0xfff8], ax
53 mov al, byte [ss:bp+0x14];cl-sector
54 xor ah, ah
55 shl ax, 1
56 shl ax, 1
57 and ax, 0x300
58 or ax, word [ss:bp+0xfff8]
59 mov word [ss:bp+0xfff8], ax
60 mov al, byte [ss:bp+0x14];cx
61 and al, 0x3f
62 xor ah, ah
63 mov word [ss:bp+0xfff4], ax
64 mov ax, word [ss:bp+012];dh-head
65 mov al, ah
66 xor ah, ah
67 mov word [ss:bp+0xfff6], ax
68 mov ax, word [ss:bp+0x6];bp
69 mov word [ss:bp+0xfff2], ax
70 mov ax, word [ss:bp+0x10];bx
71 mov word [ss:bp+0xfff0], ax
```

(b) Copying of Operation Arguments.

Figure 7.6: HD Driver Impl. 3-4

severe and might be acceptable in most cases. In our model we assume that eventually applications will also behave correctly and provide legal values, without causing repeated restarts.

The code in Figure 7.7(b) (lines 79-123) validates that disk address arguments do not exceed the actual disk parameters. These parameters are read from the Bios memory. We assume they are hard-coded and thus can not be corrupted (this code actually can be shortened and simplified by comparing to fixed disk addresses.

In Figure 7.8(a) (lines 124-148), a call to the routine that actually carries the I/O instructions appears. The needed parameters are pushed (onto the verified stack). When the call returns, the reported status is checked and in case of an error we execute the failure steps as before. If successful, we continue towards return from the driver code (`ATA 4=>0`).

7.6.3 Checking Controller State

The code in Figure 7.8(b) (lines 125-199) starts the procedure that executes I/O transfer instructions by first saving (again, the assumed fixed) Bios memory area and calculating needed parameters such as the IDE channel, salving status, and controller memory mapped addresses (all these parameters are hardwired as well).

In Figure 7.9(a) (lines 200-212), the disk block size is taken as a constant and not read from the Bios memory (as appears also in the "C" code). There is also a validation for the needed operation mode of the controller. We can not allow this mode to change during the disk operation, so we actually assume this mode is fixed for the given disk (otherwise we need to repeatedly check and set the mode). Likewise the original code contained checks of the addressing mode used, which we assume to be fixed.

The code in Figure 7.9(b) (lines 213-230) resets the count of already transferred data by writing zero in the disk controller mapped area. An internal driver counter is reset as well and used later to match the transfer status with the controller (this has a notion of the second solution, i.e., making sure the controller is following the driver, and for this solution can be omitted).

In Figure 7.10(a) (lines 231-241) the disk is checked for busy status (`ATA 1=>1`); if the busy status is on we stop and return with an error. After several such errors the disk controller will get a reset which inevitably stops the busy status (`ATA 1=>0`).

7.6.4 Writing Command Parameters and Waiting for Completion

The code in Figure 7.10(b) (lines 242-283) writes to the device all command parameters (`ATA 1=>3`). Some parameter calculations which depend on the slaving status can be eliminated if no slaved disk is present.

The code in Figure 7.11(a) (lines 284-308) waits for the disk controller preparing to carry out the command (`ATA 3=>3`). Here we see the use of leases by calling the `set_lease` procedure (line 284, implementation may be found in Figure 7.13) that sets the lease in the controller to its upper limit. In every execution of the loop that appears here there is a test for the value of the lease. In a case where the lease expires the disk is reset (`ATA 3=>0`). When the controller finally returns, we check for error status and continue execution accordingly. In our model there is the assumption that eventually the disk will succeed carrying out the command. Finally, we also make sure that the device controller turned on the DRQ bit, which means that data can be now transferred to/from memory (`ATA 3=>4`).

7.6.5 Read/Write Buffer from Disk

The code in Figure 7.11(b) (lines 309-346) clears the interrupt flag (masked by calling the soft interrupt to start the disk command) so other interrupts (e.g. the timer) can be handled from now on. We need to guarantee that other applications would not interfere before the transfer has finished. Ways for ordering access to a shared resource are discussed in Chapter 6 on memory management. A lease is used here again to wait for the disk controller before every subsequent data transfer (`ATA 4=>4`).

Line 316 branches (to fixed addresses) according to the type of operation. After this line we see the read scenario, while the write instructions appear in Figure 7.12(a) the actual data transfer into main memory is done by setting the proper register values and performing the copy (line 332).

After each read there is a check whether there are more buffers to transfer, and looping again if necessary (`ATA 4=>3`).

The code in Figure 7.12(a) (lines 347-378) performs the same steps needed for the output scenario.

The code in Figure 7.12(b) (lines 379-394) makes sure that the controller interrupts are enabled for the next I/O command, before executing a return with success indication.

7.6.6 Failure and Lease Procedures

The code in Figure 7.13 (lines 395-416) contains the mentioned severe fault handler that raises general protection error to restart the system and the code of the lease procedures. This is followed b] the general procedures used to read and write arbitrary memory values, which were omitted here.

```
; check sector counter:
;// (count > 128) || (count == 0)
72 mov ax, word [ss:bp+0xffe0]
73 cmp ax, 0x80
74 jbe continue6
75 jmp failure
continue6:
76 test ax, ax
77 jnz continue7
78 jmp failure
continue7:
```

(a) Validate Sector Count Request.

```
; read disk parameters
79 mov al, byte [ss:bp+0xffdf]
80 xor ah, ah
81 mov cx, 0x1a
82 imul ax, cx
83 mov bx, ax
84 add bx, 0x14e
85 push bx
86 push word [ss:bp+0xfffa]
87 call read_word
88 add sp, 0x4
89 mov word [ss:bp+0xffe8], ax
90 mov al, byte [ss:bp+0xffdf]
91 xor ah, ah
92 mov cx, 0x1a
93 imul ax, cx
94 mov bx, ax
95 add bx, 0x14c
96 push bx
97 push word [ss:bp+0xfffa]
98 call read_word
99 add sp, 0x4
100 mov word [ss:bp+0xffe6], ax

101 mov al, byte [ss:bp+0xffdf]
102 xor ah, ah
102 mov cx, 0x1a
104 imul ax, cx
105 mov bx, ax
106 add bx, 0x150
107 push bx
108 push word [ss:bp+0xfffa]
109 call read_word
110 add sp, 0x4
111 mov word [ss:bp+0xffe4], ax

;// sanity check on cyl heads, sec
112 mov ax, word [ss:bp+0xfff8]
113 cmp ax, word [ss:bp+0xffe8]
114 jb continue8
115 jmp failure
continue8:
116 mov ax, word [ss:bp+0xfff6]
117 cmp ax, word [ss:bp+0xffe6]
118 jb continue9
119 jmp failure
continue9:
120 mov ax, word [ss:bp+0xfff4]
121 cmp ax, word [ss:bp+0xffe4]
122 jbe continue10
123 jmp failure
continue10:
```

(b) Validate Request Against Disk Parameters.

Figure 7.7: HD Driver Impl. 5-6

```
124 push word [ss:bp+0xfff0]
125 push word [ss:bp+0xfff2]
126 push word [ss:bp+0xfffe]
127 push word [ss:bp+0xfffc]
128 push word [ss:bp+0xfff4]
129 push word [ss:bp+0xfff6]
130 push word [ss:bp+0xfff8]
131 push word [ss:bp+0xffe0]
132 mov ax, word [ss:bp+0x16];ah
133 mov al, ah
134 xor ah, ah
135 shl ax, 0x4
136 push ax
137 mov al, byte [ss:bp+0xffdf]
138 push ax
139 call ata_cmd_data_in_out
140 add sp, 0x22
141 mov byte [ss:bp+0xffde], al

;// Set nb of sector transferred
;// if (status != 0)
142 mov al, byte [ss:bp+0xffde]
143 test al, al
144 jz continue12
145 jmp failure
continue12:
146 mov sp, bp
147 pop bp
148 retn
```

(a) Call to the I/O routine and return status.

```
ata_cmd_data_in_out:
149 push bp
150 mov bp, sp
151 dec sp
152 dec sp ; get ebda_seg
153 mov ax, 0xe
154 push ax
155 mov ax, 0x40
156 push ax
157 call read_word
158 add sp, 0x4
159 mov word [ss:bp+0xfffe], ax
160 add sp, 0xfff4

161 mov ax, word [ss:bp+0x4]
162 shr ax, 1
163 mov [ss:bp+0xfff7],al
164 mov ax, word [ss:bp+0x4]
165 and al, 0x1
166 mov byte [ss:bp+0xfff6], al
167 mov al, byte [ss:bp+0xfff7]
168 xor ah, ah
169 mov cl, 0x3
170 shl ax, cl
171 mov bx, ax
172 add bx, 0x124
173 push bx
174 push word [ss:bp+0xfffe]
175 call read_word
176 add sp, 0x4
177 mov word [ss:bp+0xfffc], ax

179 mov al, byte [ss:bp+0xfff7]
180 xor ah, ah
181 mov cl, 0x3
182 shl ax, cl
183 mov bx, ax
184 add bx, 0x126
185 push bx
186 push word [ss:bp+0xfffe]
187 call read_word
188 add sp, 0x4
189 mov word [ss:bp+0xfffa], ax

190 mov ax, word [ss:bp+0x4]
191 mov cx, 0x1a
192 imul ax, cx
193 mov bx, ax
194 add bx, 0x146
195 push bx
196 push word [ss:bp+0xfffe]
197 call read_byte
198 add sp, 0x4
199 mov byte [ss:bp+0xfff3], al
```

(b) Calculate Disk Parameters.

Figure 7.8: HD Driver Impl. 7-8

```
200 mov ax, 0x200
201 mov word [ss:bp+0xfff8], ax
202 mov al, byte [ss:bp+0xfff3]
203 cmp al, 0x1
204 jnz NO_ATA_MODE_PIO32_
205 mov ax, word [ss:bp+0xfff8]
206 shr ax, 1
207 shr ax, 1
208 mov word [ss:bp+0xfff8], ax
209 jmp AFTER_ATA_MODE_PIO32_
NO_ATA_MODE_PIO32_:
210 mov ax, word [ss:bp+0xfff8]
211 shr ax, 1
212 mov word [ss:bp+0xfff8], ax
AFTER_ATA_MODE_PIO32_:

;// sector will be 0 only on lba access.
```

(a) Some More Disk Parameters.

```
;// Reset count of transferred data
213 xor ax, ax
214 push ax
215 mov ax, 0x234
216 push ax
217 push word [ss:bp+0xfffe]
218 call write_word
219 add sp, 0x6

220 xor ax, ax
221 xor bx, bx
222 push bx
223 push ax
224 mov ax, 0x236
225 push ax
226 push word [ss:bp+0xfffe]
227 call write_dword
228 add sp, 0x8

;//current = 0;
229 xor al, al
230 mov byte [ss:bp+0xfff4], al
```

(b) Reset Counters of Transferred Sectors.

Figure 7.9: HD Driver Impl. 9-10

```
;//status=inb(iobase1+ATA_CB_STAT);
;//if (status&ATA_CB_STAT_BSY)return 1;
231 mov dx, word [ss:bp+0xfffc]
232 add dx, 0x7
233 in al, dx
234 mov byte [ss:bp+0xfff5],al
235 and al, 0x80
236 test al, al
237 jz DEVICE_NOT_BUSY_
238 mov ax, 0x1
239 mov sp, bp
240 pop bp
241 retn
```

(a) Check Disk Busy Status (ATA 1=>1).

```
DEVICE_NOT_BUSY_:
242 mov al, 0xa
243 mov dx, word [ss:bp+0xfffa]
244 add dx, 0x6
245 out dx, al

246 xor al, al
247 mov dx, word [ss:bp+0xfffc]
248 inc dx
249 out dx, al

250 mov ax, word [ss:bp+0x8]
251 mov dx, word [ss:bp+0xfffc]
252 inc dx
253 inc dx
254 out dx, al

255 mov ax, word [ss:bp+0xe]
256 mov dx, word [ss:bp+0xfffc]
257 add dx, 0x3
258 out dx, al

259 mov al, byte [ss:bp+0xa]
260 mov dx, word [ss:bp+0xfffc]
261 add dx, 0x4
262 out dx, al

263 mov ax, word [ss:bp+0xa]
264 mov al, ah
265 xor ah, ah
266 mov dx, word [ss:bp+0xfffc]
267 add dx, 0x5
268 out dx, al

269 mov al, byte [ss:bp+0xfff6]
270 test al, al
271 jz NO_SLAVE_
272 mov al, 0xb0
273 jmp AFTER_NO_SLAVE_
NO_SLAVE_:
274 mov al, 0xa0
AFTER_NO_SLAVE_:
275 or al, byte [ss:bp+0xc]
276 xor ah, ah
277 mov dx, word [ss:bp+0xfffc]
278 add dx, 0x6
279 out dx, al

280 mov ax, word [ss:bp+0x6]
281 mov dx, word [ss:bp+0xfffc]
282 add dx, 0x7
283 out dx, al
```

(b) Write Command Parameters to Disk Controller (ATA 1=>3).

Figure 7.10: HD Driver Impl. 11-12

```
284 call set_lease
WHILE_WAIT_FOR_DEVICE_:
285 call test_lease
;// status=inb(iobase1+ATA_CB_STAT);
;// if ( !(status & ATA_CB_STAT_BSY) )
    break;
286 mov dx, word [ss:bp+0xfffc]
287 add dx, 0x7
288 in al, dx
289 mov byte [ss:bp+0xfff5], al
290 and al, 0x80
291 test al, al
292 jnz WHILE_WAIT_FOR_DEVICE_

;check for read errors ;
293 mov al, byte [ss:bp+0xfff5]
294 and al, 0x1
295 test al, al
296 jz NO_READ_ERROR_
297 mov ax, 0x2 ; error return value
298 mov sp, bp
299 pop bp
300 retn
NO_READ_ERROR_:
301 mov al, byte [ss:bp+0xfff5]
302 and al, 0x8
303 test al, al
304 jnz NO_DRQ_NOT_SET_ERROR_
305 mov ax, 0x3 ; error return value
306 mov sp, bp
307 pop bp
308 retn
NO_DRQ_NOT_SET_ERROR_:
```

(a) Wait for Disk Controller and Check Controller (`ATA 3=>3, 3=>4, 3=>0`).

```
309 sti

310 call set_lease
while:
311 call test_lease
312 push bp
313 mov bp, sp

; check if read or write command
314 mov ax, word [ss:bp+0x16]
315 cmp al, 0x20
316 jnz write_command

317 mov di, word [ss:bp+0x26]
318 mov ax, word [ss:bp+0x24]
319 mov cx, word [ss:bp+0x8]
320 mov es, ax
321 mov dx, word [ss:bp+0xc]
; Transfer to main memory
322 rep insd
323 mov word [ss:bp+0x26], di
324 mov word [ss:bp+0x24], es
325 pop bp
326 ;//current++;

;// count–;
327 mov ax, word [ss:bp+0x8]
328 dec ax
329 mov word [ss:bp+0x8], ax
;// status=inb(iobase1+ATA_CB_STAT);
330 mov dx, word [ss:bp+0xfffc]
331 add dx, 0x7
332 in al, dx
333 mov byte [ss:bp+0xfff5], al
;// if (count == 0)
334 mov ax, word [ss:bp+0x8]
335 test ax, ax
336 jnz DIDNT_FINISH_READING
337 mov al, byte [ss:bp+0xfff5]
338 and al, 0xc9
339 cmp al, 0x40
340 jz FINISH_READING
341 jmp error_no_sector_left

DIDNT_FINISH_READING:
342 mov al, byte [ss:bp+0xfff5]
343 and al, 0xc9
344 jz after_write_command

345 jmp error_more_sector_left
FINISH_READING:
346 jmp FINISH_READING_
```

(b) Read Transfer Loop (`ATA 4=>4, 4=>3`).

Figure 7.11: HD Driver Impl. 13-14

```
write_command:
347 mov si, word [ss:bp+0x26]
348 mov ax, word [ss:bp+0x24]
349 mov cx, word [ss:bp+0x8]
350 cmp si, 0xf800
351 mov es, ax
352 mov dx, word [ss:bp+0xc]
; Transfer from main memory
353 es rep outsd
354 mov word [ss:bp+0x26], si
;after_write_command:
355 mov word [ss:bp+0x24], es
356 pop bp
;//current++;
357 mov al, byte [ss:bp+0xfff4]
358 inc ax
359 mov byte [ss:bp+0xfff4], al

;// count–;
360 mov ax, word [ss:bp+0x8]
361 dec ax
362 mov word [ss:bp+0x8], ax
;// status=inb(iobase1+ATA_CB_STAT);
363 mov dx, word [ss:bp+0xfffc]
364 add dx, 0x7
365 in al, dx
366 mov byte [ss:bp+0xfff5], al
;// if (count == 0) {
367 mov ax, word [ss:bp+0x8]
368 test ax, ax
369 jnz DIDNT_FINISH_READING_
370 mov al, byte [ss:bp+0xfff5]
371 and al, 0xe9
372 cmp al, 0x40
373 jz FINISH_READING_

DIDNT_FINISH_READING_:
374 mov al, byte [ss:bp+0xfff5]
375 and al, 0xc9

after_write_command:
376 cmp al, 0x48
377 jz while

378 jmp error_more_sector_left
FINISH_READING_:
```

(a) Write Transfer Loop (ATA 4=>4).

```
; // Enable interrupts ;
// outb(iobase2+ATA_CB_DC,
ATA_CB_DC_HD15);
379 mov al, 0x8
380 mov dx, word [ss:bp+0xfffa]
381 add dx, 0x6
382 out dx, al
383 xor ax, ax
384 mov sp, bp
385 pop bp
386 retn

error_no_sector_left:
387 mov ax, 0x6
388 mov sp, bp
389 pop bp
390 retn

error_more_sector_left:
391 mov ax, 0x7
392 mov sp, bp
393 pop bp
394 retn
```

(b) Enable Controller Interrupts Towards Return (ATA 4=>0).

Figure 7.12: HD Driver Impl. 15-16

```
failure:
395 int 0xd ; general error

396 set_lease:
397 mov ax, LEASE_MAX_VAL
398 push ax
399 mov ax, LEASE_REGISTER
400 push ax
401 push word [ss:bp+0xfffe]
402 call write_word
403 add sp, 0x6
404 retn

405 test_lease:
406 mov ax, LEASE_REGISTER
407 push ax
408 push word [ss:bp+0xfffe]
409 call read_word
410 add sp, 0x4
411 test ax, ax
412 ja LEASE_OK
413 xor ax, ax ; reset disk command
414 int 0x13
415 call failure
LEASE_OK:
416 retn
```

Figure 7.13: HD Driver Impl. 17 - Failure Handler and Lease Procedures.

Chapter 8

Stabilizing Hosts

8.1 Introduction

"Guests, like fish, begin to smell after three days" (Benjamin Franklin). A typical computer system today is composed of several self-contained components which in many cases should be isolated from one another, while sharing some of the system's resources. Some examples are processes in operating systems, Java applets executing in browsers, and several guest operating systems above virtual machine monitors (VMM). Apart from performance challenges, those settings pose security considerations. The *host* should protect not only its various *guests* from other possibly Byzantine guests [34, 46, 83], e.g. viruses, but also must protect it's own integrity in order to allow correct and continuous operation of the system [119]. Many infrastructures today are constructed with self-healing properties, or even built to be self-stabilizing.

Recovery with no utility. The fact that the system regains consistency automatically, does not guarantee that a Byzantine guest will not repeatedly drive the system to an inconsistent state from which the recovery process should be restarted. This work expands earlier self-stabilizing efforts for guaranteeing that eventually, some of the host's critical code will be executed. This ensures that eventually the host has the opportunity to execute a monitor which can enforce it's correctness in spite of the possibly existing Byzantine guests. In particular the host forces the Byzantine code not to influence the other programs' state. Finally, non-Byzantine programs will be able to get executed by the operating system, and provide their services.

Soft errors and eventual Byzantine programs. Even if we run a closed system in which all applications are examined in advance (and during runtime), still problems like soft-errors [97] or bugs that are revealed in rare cases (due to rare i/o sequence of the environment that was not tested/considered), might

lead to a situation in which a program enters an unplanned state. The execution that starts from such an unplanned state may cause corruption to other programs or to the host system itself. This emphasizes the importance of the self-stabilization property that recovers in the presence of (temporarily or constantly) Byzantine guests. Otherwise, a single temporal violation may stop the host or the guests from functioning as required.

Host-guest enforced contract. The host guarantees preservation of the guest execution as long as the guest respects the predefined rules of a contract. The host cannot thoroughly check the guest for possible Byzantine behaviors (this is equivalent to checking whether the guest halts or not). Therefor the host will force a contract, that is sufficient for achieving useful processing for itself and the guests. The rules enforced by the host can be restrictive, e.g., never write to code segments, and allocate resources only through leases.

Stabilizing trust and reputation. Upon detecting a Byzantine behavior of a guest during run time (namely, sanity checks detect a contract violation) we can not prevent the guest from being executed, since the Byzantine behavior might be caused by a transient fault. Does this mean that we must execute all guests, including the Byzantine ones, with the same amount of resources? Furthermore, when we accumulate behavior history to conclude that a guest is Byzantine, the accumulated data maybe corrupted due to a single transient fault, thus we can not totally count on the data used to accumulate the history. Instead we continuously refresh our impression on the behavior of a guest while continuing executing all guests with different amount of resources. Details of violations are continuously gathered, and the impression depends more on recent behavior history. Such a trust and reputation function rates the guests with a suspicious level, and determines the amount of resources a guest will be granted. In this calculation, recent events are given higher weight, as past event are slowly forgotten. This approach copes with corruptions in the reputation data itself, since wrong reputation fades over time.

8.1.1 Byzantine guest examples

We review here some systems with their protection mechanisms and possible ways for Byzantine guests to attack. Commodity operating systems use standard protection mechanisms [121] such as several privilege levels and address space separation enforced by hardware, e.g., an MMU. A Byzantine guest can be executed with high privilege (by an unaware user), and corrupt the system's state. Additionally the lack of hardware i/o addresses separation in today's common processor architectures enables even kernel data corruption by, say, a faulty device driver. Managed environments like Java or the .NET CLR

Mechanism	Examples	Byzantine Threats
Privileges, Address Space separation(MMU)	Commodity OSes	(i/o) Resources tampering, Separation algo. corruption
Type checking	JVM	Resource sharing Self modifying code
Emulation and Dynamic translator	Bochs, Qemu	Compiled code corruption
Hypervisor	Xen, VmWare	Rootkits, Privileged guest corruption

Table 8.1: Byzantine Threats.

[53] use various methods of type checking, resource access control and also sandboxing [146]. These mechanisms rely on the correctness of the runtime loaders and interpreters and are also sensitive to self modifying code (see e.g., [26, 56]).

Recently, there is a growing interest in virtualization techniques through virtual machine monitors. VMMs may form full emulators like Bochs [20] and Qemu [17] which interpret (almost) all of the guest's instructions (and thus can even ensure correct memory addressing). Other VMMs, like Xen [13], let the guest perform directly most of the processor instructions, especially the non-privileged ones (in [3, 56, 113, 118, 131] there is a classification of the various VMMs types). Many VMMs rely on one of the guests for performing complex operations such as i/o, and thus are vulnerable to Byzantine behavior of this special guest. Some studies, e.g., [118], show other problems in implementing virtualization above the x86 architecture [75], including some privileged instructions which are not trappable in user mode, and direct physical memory access through DMA. Recently, vendors augmented this architecture with partial corrections [108], but still not completely [59].

"Guests" that become super-hosts. Virtualized rootkits [80, 119] were recently discussed. They load the original operating system as a virtual machine, thereby enabling the rootkit to even intercept all hardware calls that are made by the guest OS. They demonstrate the relative simplicity of a Byzantine program to take full control of the host. Table 8.1 summarizes some different mechanisms and their weaknesses.

Related research towards robust hosts. Various protection mechanisms were mentioned in the previous section. Some which emphasize separation and protection are detailed in the following. In [9], a Java virtual machine is enhanced with operating system and garbage collection mechanisms (type safety and write barriers) in order to prevent, cases like, "a Java applet can generate

excessive amounts of garbage and cause a Web browser to spend all of its time collecting it". Virtual machine emulators have become tools to analyze malicious code [56]. Lately, several studies detailed ways of preventing malicious code from recognizing that it is executing as a guest [56, 59, 80, 108, 118, 119] (see also [60] for VMM usage for security, and [119] which argues against relying on a full operating system kernel as a protection and monitoring base). In addition, well known hardware manufacturers intend to introduce soon IOMMUs with isolation capability [3, 18]. Operating system based emulators or hypervisors such as UML [39] or KVM [82] are used also to analyze suspected programs. [136] uses virtualization techniques to contain errors, especially in drivers, in realtime. Methods that are based on secure boot (e.g., [6, 116, 140]) are important in order to make sure that the host gets the chance to load first, and prevent *rootkits* from fooling it by actually running the host as a guest [80]. In [127], cryptography techniques are used in order to ensure that only authorized code can be executed.

Sandboxing techniques were presented in [146] (see also [63]). Sandboxing techniques make sure that code branches are in segment (a distinct memory section which exclusively belongs to a process), and also rely on different segments for code and data. For every computed address they add validation code that traps the system, or just masks addresses to be in the segment (this is actually a sandbox). They count on dedicated registers which hold correct addresses. Overview of trust and reputation can be found, e.g., in [64, 102]. In [25] a Bayesian based approach with exponential decay is proposed.

The need for address space separation, the use of capabilities, minimal trusted base and other protection mechanisms were introduced in well known works [19, 29, 84, 109, 121]. Singularity [71, 72] achieves process isolation in software by relying on type safety, and also prevents dynamic code. Self-modifying code certification is presented in [26].

Generally, extensive theoretical research has been done towards self-stabilizing systems [38, 41, 124] and autonomic - computing/ disaster - recovery/ reliability - availability - serviceability [74, 79, 134]. However, none of the above suggest a design for a host system that can automatically recover from an arbitrary state, even in the presence of Byzantine guests that repeatedly try to corrupt the system state.

Thesis contribution. (a) Identifying the need of combined self-stabilization, and techniques for enforcing a contract over the operations of a guest. We show that only such a combination will allow (useful) recovery. (b) The introduction of stabilizing trust and reputation and the use of the level of trust as a criteria for granting resources while continuing to evaluate the trust of the guests. (c) Concepts and a proof for designing hosts and contracts. (d) A proofed running example.

Chapter Organization. Section 8.2 details the system settings and requirements. This is followed by Section 8.3 which presents a general framework for protecting against Byzantine programs. Section 8.4 presents an example of a simple Byzantine guest followed by the way a provable host implementation copes with such a Byzantine guest.

8.2 Settings and the Requirements

Error model – arbitrary transient and Byzantine faults. The system state, including the program counter, system data structures and also the program code and data in RAM, may become arbitrarily corrupted, namely, assigned any possible value. This model is an extension of a the model from Chapter 3 used till now. The main feature of the extension is the removal of the assumption that all programs are self-stabilizing (or restartable [13]) so they might exhibit Byzantine behavior forever.

Requirements. We now define the requirements which should be satisfied for a host system to be self-stabilizing in spite of a Byzantine behavior.
(r1) Guest stabilization preservation. The fact that the host system may start in an arbitrary state, and execute code of Byzantine guests, will not falsify the stabilization property of each of the non-Byzantine guests in the system.
(r2) Efficiency guarantee. Non-Byzantine guests will eventually get the needed resources in order to supply their intended services.

Note that both (r1) and (r2) implicity require that a program that shares resources with others, will not block or will be blocked, outside of acceptable limits, due to this sharing (although in the worst case, due to the use of leases combined with a reputation system, resource will eventually be granted).

8.3 Concepts for Fighting the Byzantines

By combining techniques like secure booting, contract verification and enforcement together with self-stabilization we can protect a system against Byzantine guests in a provable way.

- **Secure booting** ensures that there is a minimal trusted computing base which runs programs and monitors.
- **Offline Byzantine behavior detectors** use code verification techniques, analyzing a program offline and looking for possible breaks of contracts.

- **Runtime anti-Byzantine enforcers** insert additional instructions in the executable for online sanity checks to enforce contract properties during a program execution.
- **Stabilizing trust and reputation** for determining the amount of resources a guest will be granted.
- **Self-stabilization** of these mechanisms and their composition [23, 41, 42] ensures that the system is eventually protected and functioning.

Secure booting is achieved through standard hardware based mechanisms (e.g., [6, 116, 140]). These are essential in order to guarantee that a Byzantine guest is not loaded first.

The system should be augmented with a detector framework which executes one or more upfront offline Byzantine detector plug-ins. A detector is built to enforce some aspect of a contract with a guest, and must be provable to perform its action completely and within acceptable time limits. These detectors scan the program code in advance for particular violations of the contract that are easy to check, and in case the scan reveals a Byzantine guest, this guest will not be loaded at all.

A program that passes the first check is augmented with sanity checks and access restrictions in sensitive code parts, where execution might do harm. The augmented code does not change the program semantics (up to stuttering) as long as the guest respect the contract. Upon detection of a violation in runtime, an enforcer can reload the program code and also update the trust and reputation level. An example for such an enforcer is one that enforces that segments used by the program are not changeable (meaning that self-modifying code is forbidden according to a contract). Runtime sanity checks, look for possible instruction sequences to make sure they do not violate the contract. Note that due to transient faults (that are rare by nature), a target address may change right after a sanity check, causing the system later to start a convergence stage as a self-stabilizing system should. In the case of a Byzantine program, the harm is prevented, although the detection and reloading will occur again and again (the trust and reputation record of a guest will limit the amount of processing used for this particular guest). In case the program is not Byzantine, the reload-of-code procedure will ensure correct behavior after which the trust and reputation will reach the maximal possible level.

Stabilizing trust and reputation can be achieved by using methods which favor recent events over past events. One example is [25] which combines a Bayesian approach with exponential decay. In such ways, trusted guests get more resources overtime, while suspected guests are not totally blocked and get chance to “shun evil and do good”. Such approaches also cope with transient (fault) corruptions in the reputation data, since wrong reputation fades over time.

Theorem 8.3.1. *There exists a self-stabilizing host that can fulfill (r1) stabilization preservation and (r2) efficiency guarantees for guests.*

Proof. We list the mechanisms we use and the properties we establish by them. (a) The host is built above a self-stabilizing hardware ([44]) which guarantees eventually correct operation of the hardware from any state. (b) A self-stabilizing host operating system which is guaranteed to periodically run some boot-code loaded in a secure way [6, 116, 140], without being subverted ([119]) (c) This trusted operating system guarantees eventual execution of all runnable processes including the contract offline detectors. (d) Code is being refreshed periodically (like in Chapters 5 and 6), so Byzantine or wrong behavior caused by transient faults to code segments are eventually fixed. (e) Contract properties are asserted by online enforcers. (f) Self-stabilizing programs might be supplied with a list of "initial" safe states. In such a case when recognizing Byzantine behavior, apart from preventing this behavior and refreshing the code, the closest state (using Hamming distance or some other metric) can be applied to the program. (g) All resource allocations are granted using leases with a self-stabilizing manager, as demonstrated in Section 6.5 about dynamic memory, ensuring that resource allocations are eventually fair. The contract detectors and enforcers check also for behavior which violates the leasing rules. Resource are leased to a guest according to its trust and reputation level. (h) System calls and traps are also leased (again, according to the trust and reputation level), so a Byzantine guest is limited in the number of times it can cause long delay due to system calls. (i) Non-Byzantine programs are stabilizing in spite of faults and Byzantine behavior. (j) The interaction between those programs and other programs or devices is stabilizing too (Chapter 7). (k) The stabilization process of one program only affects the state of this program and does not affect other programs. Thus, stabilization preservation and efficiency guarantee is achieved for guests. □

The implementations presented next, add sanity checks to branches and memory accesses, ensure correct use of leased resources, and enforce allowed patterns of out of memory accesses.

8.4 Host Implementation Example

Till this Chapter, guest separation was achieved by using the segmentation mechanism of the Pentium processor [75], without MMU hardware protection. Additionally, we assumed that the code of the programs is hardwired and correct, thus a program does not contain instructions which affect the state of the other programs (including the system). When we introduce programs with arbitrary code, other programs, even the host/operating system itself, may be

```
1  mov ax, 0x8010
2  mov ds, ax
3  mov word [0x292], 3
```

Figure 8.1: Byzantine Code.

corrupted. In the current work we have implemented a prototype of a simple host that satisfies requirements (r1) and (r2) above the mentioned system.

To demonstrate the possible corruption of the system designed, we show an example of a threat in a program that accesses the operating system's segment and changes the scheduler state. The scheduler state is changed so that this program will be scheduled (again and again) instead of other guests. Figure 8.1 shows an example of such a 16-bit x86 assembly code. Lines 1-2 change the data segment pointer to the system's segment. Then, line 3 changes the process pointer contents to a value which will cause re-scheduling of this program.

One can argue that address-space separation, like found in commodity operating system kernels, can prevent this behavior. But if a Byzantine program manages to operate in a privileged mode, even once due to a transient fault, the separation algorithm itself might be subverted, followed by the above malicious behavior. Next we will describe our settings in order to show a provable solution.

To demonstrate these ideas we show: (a) an example containing added code that enforces memory access within a program's data segments (sandboxing). (b) Accessing shared resources through leases. (c) A prototype of a detector that performs offline verification that out of segment accesses are according to a list of known patterns allowed by a contract. (d) An example of stabilizing trust and reputation evaluation according to online sanity checks.

The suggested solution uses an architecture in which some code is read-only (Harvard model). A non-maskable interrupt (NMI) is generated by a simple self-stabilizing watchdog. Thus, the hardware triggers a periodic execution of the host monitoring (detectors) code. This architecture also guarantees that the monitoring code gets enough time to complete. A detector searches the code of every program to make sure it does not contain code that changes segments outside the scope of the program. Computed addresses are enforced to be within limits, by inserting sanity checks. The correct use of leased resources is also enforced during runtime. Additionally, from time to time the host refreshes the code of all guests (including sanity checks insertions), say from a CD-ROM, to allow self-stabilization of programs following a code refresh.

(a) Figure 8.2 lines 1-2 demonstrates calculation of a segment selector value, as opposed to Figure 8.1 in which the address is fixed. Then, lines 3-6 are a sanity check added by the runtime anti-Byzantine enforcer. First the calculated address is validated to be in range (in this example it must have some fixed

```
1  mov ax, bx ;computed address
2  mov ds, ax
// added sanity check
3  xor ax, SEGMENT_MASK
4  jz AfterSanityCheck
5  call Increase-Bad-Reputation
6  mov ax, FIXED_SEGMENT
AfterSanityCheck:
...
```

Figure 8.2: Memory Access Enforcer.

```
1  call MM_Alloc
After_MM_Alloc:
2  cmp fs, 0
3  jz TryLater
...
```

Figure 8.3: Shared Resource Access.

value), in case of a violation detection (line 3) the Increase-Bad-Reputation procedure is called to record the violation (see (d) below). Then in line 6, the correct address is enforced. Alternatively, a monitor could start actions of reloading code and data in case of detecting such a wrong access.

(b) Figure 8.3 presents the way a program uses a shared resource, in this case the dynamic memory heap. The contract is that all accesses to segments in this memory area must happen only through the segment selector register **fs**. Additionally, in order for this access to be leased, a program is not allowed to load a value in this register, but instead asks the system for a leased allocation (line 1). After this allocation request, and before every dynamic memory access, the program must check that the lease is still in effect, by checking the value in **fs** (lines 2-3). In case the allocation failed or expired, the value will be 0. Detectors and enforcers check that the use of shared resources is done according to this contract (Chapter 6) and penalize the program in case of irregular use detection.

(c) The Pentium's operation code for moving a value into one of the 16-bit segment selector registers, is **8e**. Thus, the offline detector searches for commands starting with this code (assuming for simplicity that this is the only possible way). Note that the Pentium has variable length operations so in order to detect beginnings of operations we need to use disassembly techniques. Additionally, the mentioned operation code is sometimes used in legitimate ways, e.g. for accessing dynamic data segments. A possible solution is allowing known fixed patterns of access to other segments. An example pattern ap-

```
1  mov ax, VIDEO_SEGMENT
2  mov es, ax
```

Figure 8.4: An Allowed Pattern.

```
BYZANTINE-DETECTOR(process_entry, legal_patterns)
1  for each instruction_code(ic) in process_entry.code_segment
2  do if ic starts_with "8e"
3        then for each pattern in leagal_patterns
4                 do if pattern precedes ic
5                       then continue_main_loop
6                 process_entry.byzantine ← true
7                 return
```

Figure 8.5: Out of Segment Access Detector

pears in Figure 8.4, where the program is accessing the video segment, which is needed for screen output. The **es** segment register is loaded in line 2 by the allowed value that is computed in line 1. In this case the **8e** op-code is preceded by the sequence **b8 00 b8** which is considered valid. Figure 8.5 presents the algorithm of the segment access detector. This detector is executed before loading the guest program. It scans a program's code segment for the **8e** code. When found, it verifies that it is preceded by one of the allowed patterns, otherwise the program is considered as one that does not respect the contract and therefore an upfront Byzantine program.

(d) Upon finding an online contract violation through performing the enforced sanity checks, the violation is recorded in the reputation history record (Figure 8.6). This record maybe kept as part of a *process entry* in the system's process table. Every predefine period of time (or steps) the system updates this record, as seen in in the Decay-Reputation procedure in Figure 8.6. The entries in the record are shifted in a way that the oldest entry is removed, thus implementing the needed decay and stabilization. This updated record is used for evaluating the trust and reputation level of the relevant guest and granting resources in accordance.

Correctness proof. We will now prove that the system augmented with the above added detector and enforcers is self-stabilizing and fulfills our requirements in spite of possible Byzantine programs.

Lemma 8.4.1. *In every infinite system execution E, the program counter register contains the address of the detector procedure's first instruction infinitely often. Additionally, the detector code is executed completely.*

Proof. The processor eventually reaches an NMI state in which the NMI connector is set and the NMI counter contains 0. This means that the next operation

```
INCREASE-BAD-REPUTATION(process_entry)
1  process_entry.reputation[0] &= BAD_REPUTATION_BIT
2  process_entry.reputation[0] ≪ 1  ▷ Shift left.
3  return

DECAY-REPUTATION(process_entry)
1  for i in (MAX_HISTORY − 1) .. 1
2  do process_entry.reputation[i] ← process_entry.reputation[i − 1]
3  return process_entry.reputation
```

Figure 8.6: Update Trust and Reputation – Increase and Decay

that will be executed is the first operation of the NMI handler procedure. The system is configured to execute the scheduler and the detector as part of the NMI handler. Since the code of the detector is fixed in ROM, it remains unchanged. During the NMI handler execution, interrupts are not served, thus this detector can not be interrupted. Additionally, the algorithm complexity is linear in the program code size (verifying code and adding sanity checks), which enables the code to be executed completely. In case of corruption that leads to self looping, the bounded time for executing the NMI handler will cause refreshing of the code and restarting. □

Lemma 8.4.2. *A Byzantine guest will be either detected and be confined offline by the out-of-segment access detector or be prevented from unauthorized segment accesses and lease violations by the sanity checks.*

Proof. A Byzantine program, in our settings, is one which accesses unauthorized segments or leased resources. By Lemma 8.4.1 the detector is guaranteed to run completely. Since all program code is repeatedly augmented with sanity checks, any unauthorized access or use will be detected and prevented through decreasing allocation of the different system resource to a misbehaving guest. □

Lemma 8.4.3. *In every infinite execution E, stabilization preservation eventually holds.*

Proof. Stabilization preservation is achieved for a non-Byzantine guest by the design of the system (stabilization of the scheduler, program loading and refreshing and resource managers). The only threat left is a Byzantine process. By Lemma 8.4.1 all code will eventually be verified. By Lemma 8.4.2, Byzantine execution of a program, which corrupts the state of another program, will be eventually detected, restricted and granted less resources. □

Lemma 8.4.4. *In every infinite execution E, efficiency guarantee is achieved.*

Proof. Similar to Lemma 8.4.3 and by the design of the self-stabilizing scheduler, which keeps incrementing its process pointer, thus all non-Byzantine processes which are marked for running, are executed infinitely often. Additionally, other shared resources like dynamic memory, are lease based and are given by the self-stabilizing resource manager. The trust and reputation function is also stabilizing, and enables resource usage in spite of transient faults. The added sanity checks are implemented using simple masking and comparison operations, thus they do not have critical impact on program running times. The offline detector procedure is dependent only on program size and the fixed number of allowed instruction patterns. □

Corollary 8.4.5. *The system augmented with the out-of-segment access detector and online enforcers satisfies (r1) guest stabilization preservation and (r2) efficiency guarantee in spite of Byzantine programs.*

Performance issues. The timing of the execution of code refreshing (and offline detectors) can be tuned according to the expected rate of soft-error corruptions to code. This processes do not have a great impact on the program execution performance, since the frequency of the checks may be balanced against the desired recovery speed.

One could suggest a performance gain by having an auxiliary processor (one core of a multi-core) for performing a repeated contract verification on the loaded code. However, a Byzantine guest might fool the auxiliary processor, say, by changing the sanity checks to be correct whenever the auxiliary processor is checking them. Still we can use such a processor for most of the cases to speed the indication on soft errors and to trigger code refreshing.

8.5 Concluding Remarks

In this chapter we presented an approach to use self-stabilizing reputation in order to gain efficient performance. We believe that self-stabilizing host systems that use stabilizing reputation are a key technology which can cope with Byzantine behavior in critical computing system. The source code examples can also be found in [132].

Chapter 9

Concluding Remarks

This thesis presented building blocks towards the vision of implementing self-stabilization in practical (and critical) systems. In particular, the first implementation of a self-stabilizing operating system is presented.

Performance issues. The performance of such an operating system can be tuned according to the time period required and chosen for recovery. The timing of the execution of code refreshing can be tuned according to the expected rate of soft-error corruptions to code. The various consistency checks also do not have a great impact on the program execution performance, since the frequency of the checks may be balanced against the desired recovery speed.

For example, in the implementation of Section 5.3, the scheduler code contains 68 lines of code of which only 6 lines (less than 10%) are directly concerned with consistency checks. In other parts of the implementation the ratio is even smaller. As mentioned, the execution frequency can also be tuned. Additionally, some of the checks are only optimization since the applications are self-stabilizing too. In our example, two out of three checks can be ignored assuming the applications are given enough time to stabilize.

Tests comparing the presented code against the same code excluding checks required for self-stabilization, were performed. The tests included the same task (output of text to the screen, as in [132]) being run by increased number of processes. There was an almost fixed and negligible time difference in the executable between the two cases (around 0.25%). Test results are available at [132], as well.

One could suggest a performance gain by having an auxiliary processor (one core of a multi-core) for performing code refreshes or executing the scheduler task with its sanity checks. Although in some rare scenarios the auxiliary processor might be "fooled", say, by a program accidently changing values to be correct whenever the auxiliary processor is checking them. Still we can use such a processor for most of the cases to speed state corrections and to trigger code refreshing.

We have run the presented systems using the BOCHS [20] simulator. During some of the executions we completely changed the contents of the RAM and observed that stabilization was achieved. Namely, the processor eventually continues to execute the correct code of the operating system and applications.

Application to real-time systems: since the implementation is written in assembly without relying on other execution environments, we can compute deadlines for the system operations. Once the system stabilizes those deadline will be respected.

We started with the black-box solutions in Chapter 4. Then Chapters 5, 6 and 7 described the tailored solution according to main kernel components. In Chapter 8 protection against Byzantine programs was added.

The usage and usefulness of such a system in critical and remote systems cannot be over emphasized. For example entire years of work maybe lost when the operating system of an expensive complicated device (e.g., an autonomous spaceship) may reach an arbitrary state (say, due to soft errors) and be lost forever (say, on Mars). The controllers of a critical facility (e.g., a nuclear reactor or even a car) may experience an unexpected fault (e.g., an electrical spike) that will cause it to reach an unexpected state, from which the system will never recover, therein leading to harmful results.

We make a step to use the well founded self-stabilization paradigm as part of the design of on-going critical systems. This effort should lead to robust systems and hence better service for us all.

Bibliography

[1] V. Abrossimov, F. Herrmann, J. Christophe Hugly, F. Ruget, E. Pouyoul, M. Tombroff. "Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology", *Technical Report CSI-T4-96-34*, Chorous Sustems, Inc., August, 1996.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. "MACH: A New Kernel Foundation for UNIX Development", *Proceedings of the USENIX Summer Conference*, Atlants, GA, 1986.

[3] K. Adams, O. Agesen. "A Comparison of Software and Hardware Techniques for x86 Virtualization", *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, CA., 2006.

[4] C. Allison. "Wanted: An Application Aware Checkpointing Service", *Proceedings of 6th ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, Germany 1994.

[5] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. Da Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, J. Xenidis. "Enabling Autonomic System Software with Hot-Swapping", *IBM Systems Journal*, Vol. 42, No. 1, pg 60-76, 2003.

[6] W. A. Arbaugh, D. J. Farber, J. M. Smith. "A secure and reliable bootstrap architecture", *In Proceedings of 1997 IEEE Symposium on Computer Security and Privacy*, 1997.

[7] Francois Armand. "ChorusOS Features and Architecture overview", Sun Technical Report, December 2001. `http://www.jaluna.com/developer/papers/COSDESPERF.pdf`.

[8] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr. "Basic concepts of dependable and secure computing", *IEEE Transactions on Dependable and Secure Computing*, January 2004.

[9] G. Back, W. H. Hsieh, J. Lepreau. "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java", *In Proc. 4 th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2000.

[10] M. Baker, M. Sullivan. "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment", *Proceedings of the Summer 1992 USENIX Conference*, Texas, June 1992

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, A. Ustuner. "Thorough static analysis of device drivers", *Proceedings of EuroSys'06: European Systems Conference*, 2006.

[12] T. Ball, S.K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis", *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*, Portland, OR, 2002.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. "Xen and the Art of Virtualization", *Proceedings of the nineteenth ACM symposium on Operating systems principles*, Bolton Landing, NY, USA, 2003.

[14] P. Barham, B. Dragovich, K. Fraser, S. Hand, A. Ho, I. Pratt. "Safe Hardware Access with the Xen Virtual Machine Monitor", *1st Workshop on Operating System and Architectural Support for On-Demand IT Infrastructure*, May 2004.

[15] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. G. R. Horst, R. Jardine, D. Lenoski, D. McGuire. "Fault Tolerance in Tandem Computer Systems". *Technical Report TR-90.5*, Tandem, 1990.

[16] L. A. Belady, R. P. Parmelee, C. A. Scalzi. "The IBM History of Memory Management Technology", *IBM Journal of Research and Development 25(5)*, pp. 491-504, 1981.

[17] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator", *Proc. of USENIX Annual Technical Conference*, FREENIX Track, 2005.

[18] M. Ben-Yehuda, J. Xenidis, M. Mostrows, K. Rister, A. Bruemmer, L. Van Doorn. "The Price of Safety: Evaluating IOMMU Performance", *The 2007 Ottawa Linux Symposium (OLS)*, 2007.

[19] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, C. Chambers. "Extensibility, Safety, and Performance in the SPIN Operating System", *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Colorado, December 1995.

[20] Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net`.

[21] A. Borg, W. Blau, W. Graetsch, F. Herrmann, W. Oberle. "Fault Tolerance Under UNIX", *ACM Transactions on Computer Systems*, 7(1):1-24, February 1989.

[22] O. Brukman, S. Dolev, Y. Haviv, L. Lahiani, R. Kat, E. M. Schiller, N. Tzachar, R. Yagel. "Self-Stabilizing from Theory to Practice". *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, No. 94, Febuary 2008

[23] O. Brukman, S. Dolev, Y. Haviv, R. Yagel. "Self-Stabilization as a Foundation for Autonomic Computing" . *Proceedings of the Second International Conference on Availability, Reliability and Security, Workshop on Foundations of Fault-tolerant Distributed Computing (FOFDC)*, Vienna, Austria, April 2007.

[24] O. Brukman, S. Dolev, H. Kolodner. "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software", *Proceedings of IEEE International Conference on Software-Science Technology & Engineering*, (SwSTE03), Israel, 2003.

[25] S. Buchegger, J.-Y. Le Boudec. "A Robust Reputation System for Mobile Ad-hoc Networks". *Technical Report IC/2003/50*, EPFL-IC-LCA, 2003.

[26] H. Cai, Z. Shao, A. Vaynberg. "Certified Self-Modifying Code", *Proceedings of PLDI'07*, CA., 2007.

[27] M. Castro, B. Liskov. "Proactive Recovery in a Byzantine-Fault-Tolerant System", *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pp. 273-288, San Diego, USA, October 2000.

[28] R. H. Campbell, N. Islam, D. Raila, P. Madany. "Designing and Implementing Choices: An Object-Oriented System in C++", *Communication of the ACM*, 36(9): 117-126, September 1993.

[29] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska. "Sharing and Protection in a Single-Address-Space Operating System". *ACM Transactions on Computer Systems*, 12(4), November 1994.

[30] A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler. "An empirical study of operating systems errors", *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[31] B. Cook, A. Podelski, A. Rybalchenko. "Terminator: Beyond safety", *CAV06: Computer-Aided Verification (2006), vol. 4144 of LNCS*, Springer-Verlag, pp. 415418

[32] F. J. Corbato, "Turing Lecture Paper: On Building Systems That Will Fail". `http://larch-www.lcs.mit.edu:8001/∼corbato/turing91/`.

[33] R. C. Daley , J. B. Dennis. "Virtual memory, processes, and sharing in Multics", *Proceedings of the first ACM symposium on Operating System Principles*, p.12.1-12.8, January 1967, Gatlinburg, TN.

[34] A. Daliot, D. Dolev. "Self-stabilizing Byzantine Agreement", *Proc. of Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*, Colorado, 2006.

[35] B. Demsky and M. Rinard. "Automatic Detection and Repair of Errors in Data Structures", *Technical Report MIT-LCS-TR-875*, MIT, 2002. `http://bdemsky.mit.edu/MIT-LCS-TR-875.pdf`

[36] P. J. Denning. "Fault Tolerant Operating Systems", *ACM Computing Surveys (CSUR)*, v.8 n.4, p.359-389, Dec. 1976

[37] E. W. Dijkstra. "Cooperating Sequential Processes", *Chap. in The origin of concurrent programming: from semaphores to remote procedure calls book contents*, Springer-Verlag, USA, 2002

[38] E. W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control", *Communications of the ACM,* Vol. 17, No. 11, pp. 643-644, 1974.

[39] J. Dike. "A User-mode Port of the Linux Kernel", *In 5th Annual Linux Showcase and Conference*, Oakland, California, 2001.

[40] E.W. Dijkstra. "My recollections of operating system design", `http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF`, April 2001.

[41] S. Dolev. *Self-Stabilization*, The MIT Press, Cambridge, 2000.

[42] S. Dolev, T. Herman. "Parallel Composition of Stabilizing Algorithms" *Proceedings of the 4th Workshop on Self-Stabilizing Systems*, (WSS 1999), pp. 25-33, 1999.

[43] S. Dolev, Y. Haviv. "Self-Stabilizing Microprocessor: Analyzing and Overcoming Soft Errors", *IEEE Trans. on Computers 55(4)*, 2006. Also at: *17th International Conference on Architecture of Computing Systems (ARCS04)*, 2004.

[44] S. Dolev, Y. Haviv. "Stabilization Enabling Technology". *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pp. 1-15, Texas, November 2006.

[45] S. Dolev, R. Kat. "Self-Stabilizing Distributed File Systems", *International Workshop on Self-Repairing and Self-Configurable Distributed Systems*, (RCDS 2002), pp. 384-389, 2002.

[46] S. Dolev, J. L. Welch. "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults", *Proceedings of the Second Workshop on Self-Stabilizing Systems*, (WSS 1995), pp. 9.1-9.12, 1995. Also in *Proceedings of the 14th Annual ACM Symp. on Principles of Distributed Computing*, (PODC 1995), pp. 256, 1995.

[47] S. Dolev, R. Yagel. "Toward Self-Stabilizing Operating Systems", *Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems* (SAACS04,DEXA), pp. 684-688, Zaragoza, Spain, August 2004.

[48] S. Dolev, R. Yagel. "Memory Management for Self-Stabilizing Operating Systems", *Proceedings of the 7th Symposium on Self Stabilizing Systems*, Barcelona, Spain, October 2005. Also in *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 2006.

[49] S. Dolev, R. Yagel. "Self-Stabilizing Device Drivers", *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pp. 276-289, Dallas, Texas, USA, November 2006.

[50] S. Dolev, R. Yagel. "Stabilizing Trust and Reputation for Self-Stabilizing Efficient Hosts in Spite of Byzantine Guests". *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, France, November 2007.

[51] J. R. Douceur, W. J. Bolosky. "Progress-based regulation of low-importance processes", *Proceedings of 17th ACM Symposium on Operating Systems Principles*, pages 247-60, ACM Press, December 1999.

[52] W. R. Dunn. "Designing Safety-Critical Computer Systems", *IEEE Computer*, 40-56, November 2003.

[53] ECMA International. *ECMA-335 Common Language Infrastructure (CLI)*, 4th Edition, Technical Report, 2006.

[54] D.R. Engler, M.F. Kaashoek. "Exterminate All Operating System Abstractions ", *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995. IEEE Computer Society.

[55] D.R. Engler, M.F. Kaashoek, J. W. O'Toole Jr. "The operating system kernel as a secure programmable machine", *Proceedings of 6th ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, Germany 1994.

[56] P. Ferrie. "Attacks on Virtual Machine Emulators", Symantec Advanced Threat Research, `http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

[57] B. Ford, K. Van Maren, J. Lepreau, S. Clawson, B. Robinson, J. Turner. "The Flux OS Toolkit: Reusable Components for OS Implementation", *Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[58] A. Fox, D. Patterson. "Self-Repairing Computers", *Scientific American*, June, 2003

[59] T. Garfinkel, K. Adams, A. Warfield, J. Franklin. "Compatibility Is Not Transparency: VMM Detection Myths and Realities", *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, 2007.

[60] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh. "Terra: A virtual machine-based platform for trusted computing", *In Proceedings of SOSP 2003*, Oct. 2003.

[61] M. Gien. "Micro-Kernel Architecture: Key to Modern Operating System Design". *UNIX Review*, November 1990.

[62] C. Gray, D. Cheriton. "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency". *Proceedings of the 12th ACM Symposium on Operating System Principles, pp. 202-10, 1989.*

[63] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers. “Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2”, *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[64] R. Guha, R. Kumar, P. Raghavani, A. Tomkins. “Propagation of trust and distrust”, *In Proceedings of the 13th International World Wide Web conference (WWW)*, 2004.

[65] M. Herdieckerhoff, F. Ruget. “Matching operating systems to application needs: a case study”. *Proceedings of 6th ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, Germany 1994.

[66] T. Herman, T. Masuzawa. “Available stabilizing heaps”, *Inf. Process. Lett. 77(2-4)*, 2001.

[67] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. “System Architecture Directions for Networked Sensors”, ASPLOS 2000.

[68] C. A. R. Hoare. “Monitors: An operating system structuring concept”, *Communications of the ACM*, 17(10):549-557, October 1974.

[69] Y. Hong, D. Chen, L. Li, K. S. Trivedi. “Closed Loop Design for Software Rejuvenation”, *Workshop on Self-Healing, Adaptive, and Self-Managed Systems (SH AMAN)*, 2002.

[70] Y. Huang, P. E. Chung, C. Kintala, C. Wang, D. Liang. “NT-SwiFT: Software Implemented Fault Tolerance on Windows NT”, *Proceedings of the 2nd USENIX Windows NT Symposium*. Seattle, Washington, August 34, 1998.

[71] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, T. Wobber. “Sealing OS Processes to Improve Dependability and Safety”, *Proceedings of EuroSys2007*, Lisbon, Portugal, March 2007.

[72] G. Hunt, J. Larus. “Singularity: Rethinking the Software Stack”, *Operating Systems Review*, Vol. 41, Iss. 2, April 2007.

[73] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, B. Zill. “An Overview of the Singularity Project”, *Microsoft Research Technical Report MSR-TR-2005-135*, Microsoft Corporation, Redmond, WA, October 2005.

[74] Intel Corporation. “Reliability, Availability, and Serviceability for the Always-on Enterprise, The Enhanced RAS Capabilities of Intel Processor-based Server Platforms Simplify 24 x7 Business Solutions”, Technology@Intel Magazine, August, 2005. http://www.intel.com/technology/magazine/Computing/Intel_RAS_WP_0805.pdf

[75] Intel Corporation. “The IA-32 Intel Architecture Software Developer’s Manual”, http://developer.intel.com/products/ processor/manuals/index.htm, 2007.

[76] Jaluna-2/RT Technical Features, http://www.jaluna.com/proserv/ jaluna2-rt.features.html.

[77] Jaluna. “C5 1.0 Features and Architecture Overview”, *Jaluna Technical Paper JL/TR-02-51.1*. October 2002.

[78] G. Kaiser, J. Parekh, P. Gross, G. Valetto. “Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems”. *5th Annual International Active Middleware Workshop*, pp. 22-30, June 2003.

[79] J. O. Kephart, D. M. Chess. “The Vision of Autonomic Computing”, *IEEE Computer*, 41-50, January, 2003. See also http://www.research.ibm.com/autonomic.

[80] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, J. R. Lorch. "SubVirt: Implementing malware with virtual machines", *IEEE Symposium on Security and Privacy*, May 2006.

[81] M. Kistler, P. Shivakumar, L. Alvisi, D. Burger, and S. Keckler. "Modeling the effect of technology trends on the soft error rate of combinational logic". In *ICDSN*, volume 72 of *LNCS*, pages 216–226, 2002.

[82] *KVM: Kernel-based Virtual Machine for Linux*, `http://kvm.qumranet.com/`

[83] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem", *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401, 1982.

[84] B.W. Lampson. "Protection", *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971. Reprinted in *ACM Operating Systems Review*, January, 1974.

[85] B. W. Lampson. "How to build a highly available system using consensus", *Distributed Algorithms,* LNCS *1151*, 1996.

[86] B. W. Lampson, H.E. Sturigs. "Reflections on an Operating System Design", *Communications of the ACM*, 19(5):251-265, May

[87] C. R. Landau. "The checkpoint mechanism in KeyKOS", *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pp. 86-91, September, 1992. see also `http://www.eros-os.org/design-notes/Checkpoint.html`.

[88] B. Leslie, G. Heiser. "Towards untrusted device drivers", *Technical Report UNSW-CSE-TR-0303, School of Computer Science and Engineering*, 2003.

[89] J. LeVasseur, V. Uhlig. "A Sledgehammer Approach to Reuse of Legacy Device Drivers", *Proceedings of the 11th ACM SIGOPS European Workshop*, Belgium, 2004.

[90] J. LeVasseur, V. Uhlig, J. Stoess, S. Götz. "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, 2004

[91] P. Levis, D. Culler. "Mate: A Tiny Virtual Machine for Sensor Networks", *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002.

[92] T. A. Linden. "Operating System Structures to Support Security and Reliable Software", *ACM Computing Surveys (CSUR)*, v.8 n.4, p.409-445, Dec. 1976

[93] Linux Online. `http://www.linux.org`.

[94] H. J. Löeser, F. Mehnert, L. Reuther, M. Pohlack, A. Warg. "An I/O Architecture for Mikrokernel-Based Operating Systems", *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, 2004

[95] D. E. Lowell, S. Chandra, P. M. Chen. "Exploring failure transparency and the limits of generic recovery", *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.

[96] A. Mahmood, E.J. McCluskey., "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Trans. on Computers*, vol. 37, no. 2, 160-174, February 1988.

[97] R. Mastipuram, E. C. Wee. "Soft errors' impact on system reliability". *Voice of Electronics Engineer*, `http://www.edn.com/article/CA454636.html`, 2004

[98] M. Mactaggart. “Thinking outside the box”, *Application Development Advisor*. ADA Communications Ltd., June 2002.

[99] B. Meyer. “Applying “design by contract” ”, *IEEE Computer*, 40-51, October 1992.

[100] B. Meyer. “Object-Oriented Software Construction”, second edition. Prentice Hall. 1997.

[101] Microsoft. “Windows XP/Office XP Feature Overview”. `http://www.microsoft.com/windowsxp/pro/evaluation/overviews/windowsxpofficexp.asp`.

[102] L. Mui. “Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks”, *Ph.D. thesis*, Massachusetts Institute of Technology, Cambridge, MA, 2002.

[103] H. Munz. “LP-VxWin VxWorks Together with Windows on the same PC”, *Real-Time Magazine 97Q2 on RTOS Update (part1)*, 1997, `http://www.realtime-info.be/magazine/97q2/1997q2_p047.pdf`.

[104] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, A. Ulbrich. ”Self-stabilizing publish/subscribe systems: Algorithms and evaluation”. *Proceedings of the 11th European Conference on Parallel Processing (Euro-Par 2005), (*LNCS *3648)*, 2005.

[105] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, T. Austin. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”. *Proc. 36th Annual Int'l Symp. on Microarchitecture (MICRO)*, 2003.

[106] S, J, Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, H. van Staveren. “Amoeba: a distributed operating system for the 1990s“, *IEEE computer*, 23(5):44-53, May 1990.

[107] Nasm. The Netwide Assembler. `http://nasm.sourceforge.net`.

[108] G. Neiger, A. Santony, F. Leung, D. Rogers, R. Uhlig. “Virtualization Technology: Hardware Support for Efficient Processor Virtualization”, *Intel Technology Journal*, Volume 10, Issue 3, Augost 2006.

[109] P.G. Neumann. “Computer-Related Risks”, Addison-Wesley, Menlo Park, California, 1995.

[110] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, L. Robinson. “A provably secure operating system: The system, its applications, and proofs”, *Technical Report CSL-116, SRI International*, 1980.

[111] NMI. `http://slacksite.com/slackware/nmi.html`

[112] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. “Recovery Oriented Computing(ROC): Motivation, definition, techniques and case studies”, UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002.

[113] G. J. Popek, R. P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM 17 (7)*: 412 421, 1974.

[114] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau. “IRON File Systems”, *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)* Brighton, UK, October 2005.

[115] QNX Software Systems. Realtime operating system software. `http://www.qnx.com/`.

[116] E. Ray, E. E. Schultz. "An early look at Windows Vista security", *Computer Fraud & Security Volume 2007, Issue 1*, 2007.

[117] Red Hat, Inc. "eCos: Embedded Configurable Operating System", Version 1.3. `http://www.redhat.com/products/ecos/`.

[118] J. Robin, C. Irvine. "Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor", *Usenix annual technical conference*, 2000.

[119] J. Rutkowska. "Subvirting Vista Kernel For Fun and Profit — Part II Blue Pill", `http://www.whiteacid.org/ misc/bh2006/070_Rutkowska.pdf`, see also Red-Pill, `http://invisiblethings.org/ papers/redpill.html`.

[120] Jerome H. Saltzer. "Protection and the control of information sharing in multics", *Communications of the ACM*, v.17 n.7, p.388-402, July 1974.

[121] J.H. Saltzer, M.D. Schroeder. "The protection of information in computer systems", *Proceedings of the IEEE*, 63 (9). pp. 1268-1308, September 1975. see also: M.D. Schroeder. "Co-operation of Mutually Suspicious Subsystems in a Computer Utility", *Ph.D. dissertation*, Massachusetts Institute of Technology, Cambridge, MA, September 1972.

[122] P. Rogina, G. Wainer. "Extending RT-MINIX with Fault Tolerance Capabilities", *Proceedings of the Latin-American Conference on Informatics (CLEI)*, Merida, Venezuela, 2001.

[123] J. Saltzer, D. Reed, D. Clark. "End-to-end arguments in system design", *ACM Trans. Computer System*, 2(4), November 1984.

[124] Self-Stabilization Home Page. `http://www.selfstabilization.org`

[125] M. Seltzer, C. Small. "Self-monitoring and self-adapting operating systems", *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, May 1997.

[126] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, M. Miller. "Towards a verified, general-purpose operating system kernel". Available at `http://www.coyotos.org`, 2005.

[127] A. Sharma, S. Welch. "Preserving the integrity of enterprise platforms via an Assured eXecution Environment (AxE)", *A poster at the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006

[128] A. Shieh, D. Williams, E. Gün Sirer, F. B. Schneider. "Nexus: A New Operating System for Trustworthy Computing", *Symposium on Operating Systems Principles WIP Session*, United Kingdom, October 2005.

[129] ACM SIGOPS - The Special Interest Group for Operating Systems. "SIGOPS - How to Write an Operating System", `http://www.acm.uiuc.edu/sigops/roll_your_own/`.

[130] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts*, 6th edition. John Wiley & Sons, 2003.

[131] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson. "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", *In Proceedings of the 2007 EuroSys conference*, Lisbon, Portugal, March 2007.

[132] SOS download page. `http://www.cs.bgu.ac.il/~yagel/sos`, 2008

[133] M. Spear, T. Roeder, O. Hodson, G. Hunt, S. Levi. "Solving the Starting Problem: Device Drivers as Self-Describing Artifacts", *Proceedings of EuroSys2006*. Leuven, Belgium, April 2006.

[134] Sun Microsystems, Inc. 'Predictive Self-Healing in the Solaris™10 Operating System", *White paper*, September 2004. `http://www.sun.com/software/solaris/ds/self_healing.pdf`.

[135] M. Swift, M. Annamalai, B. N. Bershad, H. M. Levy. "Recovering Device Drivers", *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (ODSI)*, San Francisco, 2004.

[136] M. Swift, B. N. Bershad, H. M. Levy. "Improving the reliability of commodity operating systems", *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, Bolton Landing, NY, October 2003. See also: M. Swift. "Improving the Reliability of Commodity Operating Systems", *Ph.D. Dissertation*, University of Washington, 2005.

[137] T13 InterNational Committee for Information Technology Standards, *ATA Storage Interface - T13/1532D Vol. 2. Revision 4a (working drafts)*. `http://www.t13.org/#Projects`.

[138] A. S. Tanenbaum. *Modern Operating Systems*, 2nd edition. Prentice Hall, New Jersey, 2001.

[139] A. S. Tanenbaum, A. S. Woddhull. *Operating Systems Design and Implementation*, 3nd edition. Prentice Hall, New Jersey, 2006.

[140] J. D. Tygar, B. Yee. "Dyad: A system for using physically secure coprocessors" *In Proceedings of IP Workshop*, 1994.

[141] UC Berkeley lab project, "Watchdog Designs for TinyOS Motes", `http://www.cs.berkeley.edu/ iben/Classes/cs252`.

[142] K. T. Van Maren. "The Fluke Device Driver Framework", *Master's thesis*, The University of Utah, 1999.

[143] B. Venners. "The Demand for Software Quality A Conversation with Bertrand Meyer", Artima interview, October 2003. `http://www.artima.com/intv/serious.html`

[144] J. von Neumann. "First Draft of a Report on the EDVA", Moore School of Electrical Engineering, Univ. of Penn., Philadelphia, 1945, Reprinted (in part) in B. Randell. *Origins of Digital Computers: Selected Papers*, Springer-Verlag, Berlin Heidelberg, pp. 383-392, 1982.

[145] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 329–378. Princeton University Press, Princeton, MJ, 1956.

[146] R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham. "Efficient Software-based fault isolation", *Proceedings of the Sym. On Operating System Principles*, 1993

[147] J. Wilkes. "Better Mousetraps", *Proceedings of 6th ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, Germany 1994.

תקציר

בשנים האחרונות גבר העניין בתכנון מערכות מחשב עמידות בפני תקלות, בין השאר בשל התגברות הרגישות של רכיבים לשגיאות רכות (Soft Errors) [81, 97, 105]. העבודה מציגה מספר גישות לעיצוב מערכת הפעלה מתייצבת עצמית (Self-stabilizing).

התייצבות עצמית [38, 41, 124] הנה גישה אלגנטית לתכנון ובניית מערכות מחשב עמידות בפני תקלות. מערכת מחשב מוגדרת כ-"מתייצבת עצמית" אם היא מתכנסת במהלך ריצתה להתנהגות רצויה, בלא תלות במצב בו היא מאותחלת. אלגוריתמים מתייצבים עצמית מיושמים עבור פתרון בעיות מגוונות בחישוב מבוזר. עבודה זו באה לתת מענה להרצת תוכניות מתייצבות עצמית ברכיב בודד, זאת כחלק ממחסנית של כלי תשתית בנושא [22, 23].

הגישה הראשונה בעבודה זו, היא בגישת הקופסא-שחורה ומבוססת על התקנה תקופתית ואוטומטית של מערכת הפעלה קיימת ואתחולה מחדש. גישה מעודנת יותר, מתקינה מחדש את חלק הפקודות של מערכת-ההפעלה בלבד ומשתמשת בפרדיקטים על מצב המערכת (תוכנם של משתנים) כדי להבטיח שהיא אינה חורגת מהמפרט שלה [48].
מכאן עוברת העבודה לבניית מערכת הפעלה קטנה מותאמת במיוחד לצורך התייצבות עצמית. מתוארים ומפורטים המנגנונים העיקריים המקובלים בגרעין מערכת הפעלה: ניהול תהליכים [47], ניהול זיכרון [48] והתקני קלט/פלט [49]. כמו כן, מנוסחות הדרישות הנוספות מחלקים אלו.
בנוסף, מתוארים מספר מימושים של אבות-טיפוס למערכת הפעלה שכזו מעל המעבד פנטיום של אינטל [75]. לסיום, מתואר כיצד מערכת הפעלה כזו יכולה להוות את הבסיס למערכת מארחת מוגנת מפני תקיפות ביזנטיניות [50].

בעקבות ליקוי בלתי צפוי בהתקן קריטי, למשל בקר בכור גרעיני או אפילו במכונית, מערכת יכולה להימצא במצב בלתי צפוי שממנו היא אינה יכולה להתאושש. באמצעות המערכת המתוארת בעבודה, ההתקן יוכל בסופו של דבר להתייצב ובכך למנוע תוצאות הרסניות.

העבודה נעשתה בהדרכת

פרופ׳ שלומי דולב

במחלקה למדעי המחשב

בפקולטה למדעי הטבע

מערכות הפעלה
מתייצבות עצמית

מחקר לשם מילוי חלקי של הדרישות לקבלת תואר "דוקטור לפילוסופיה"

מאת

ראובן יגל

הוגש לסינאט אוניברסיטת בן גוריון בנגב

אישור המנחה ________________

אישור דיקן בית הספר ללימודי מחקר מתקדמים ע"ש קרייטמן ___________________

אלול תשס"ז ספטמבר 2007

באר שבע

מערכות הפעלה
מתייצבות עצמית

מחקר לשם מילוי חלקי של הדרישות לקבלת תואר "דוקטור לפילוסופיה"

מאת

ראובן יגל

הוגש לסינאט אוניברסיטת בן גוריון בנגב

אלול תשס"ז ספטמבר 2007

באר שבע