Self-Stabilizing Device Drivers

SHLOMI DOLEV and REUVEN YAGEL
Ben-Gurion University of the Negev

This work presents approaches for designing the input-output device management components of self-stabilizing operating systems. As an example, we demonstrate the nonstability of the ATA standard protocol for storage devices. We state the requirements that an operating system and I/O devices should satisfy in order to become self-stabilizing. Then we suggest two solutions to satisfy these requirements. The first uses leases to guarantee progress from the I/O device side. The second assumes stabilization of the I/O device, and uses snapshots to perform consistency checks. A device driver for a PC hard-disk, using the first solution, was implemented. By supplying an infrastructure for practical self-stabilizing systems, robust and dependable systems can be achieved.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Fault tolerance*; D.4.5 [**Operating Systems**]: Reliability

General Terms: Self-stabilizing systems, device driver failures, ATA interface standard

ACM Reference Format:

Dolev, S. and Yagel R., 2008. Self-stabilizing device drivers. ACM Trans. Auton. Adapt. Syst. 3, 4, Article 17 (November 2008), 29 pages. DOI = 10.1145/1452001.1452007 http://doi.acm.org/10.1145/1452001.1452007

1. INTRODUCTION

Device drivers are known to be a major cause of operating system failures [Chou et al. 2001; Swift 2005; Swift et al. 2003] 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, essential system parts are usually designed, built, verified, and tested with extra care, while often drivers are brought from the outside. The following techniques are used to deal with

This work was partially supported by Rafael, Microsoft, MFAT, IBM, NSF, STRIMM, the Rita Altura Trust Chair in Computer Sciences, and the Lynne and William Frankel Center for Computer Sciences.

An extended abstract of this work was presented at the 8th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Dallas, TX.

Authors' address: Ben Gurion University of the Negev, Beer-Sheva, 84105, Israel; email: {dolev,yagel}@cs.bgu.ac.il.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2008 ACM 1556-4665/2008/11-ART17 $5.00 DOI 10.1145/1452001.1452007 http://doi.acm.org/10.1145/1452001.1452007

these failures: (a) reducing the driver's access to system resources [Hunt et al. 2005; Tanenbaum and Woddhull 2006], (b) containment of errors in realtime through virtualization [Barham et al. 2004; Swift et al. 2003], (c) using typed languages [Hunt et al. 2005; Shapiro et al. 2005], and (d) static analysis of the drivers' code, and of their resource use [Ball and Rajamani 2002; Spear et al. 2006]. 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 [Mukherjee et al. 2003]) in device drivers accumulate and lead to undesired behavior.

Device drivers (or simply drivers) are programs that 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, for example, ATA and SCSI for disk drives. In Prabhakaran et al. [2005] it is stated that "a modern Seagate drive contains roughly 400,000 lines of code." In Tanenbaum and Woddhull [2006] it is noted that "modern disk controllers often have many megabytes of memory inside the controller."

According to Chou et al. [2001] and LeVasseur et al. [2004] about 70% of operating system code is devoted to device drivers. It is stated in Swift [2005] that "In Windows XP, for example, device drivers cause 85% of reported failures." Moreover, in Chou et al. [2001] 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." The complexity of today's I/O devices enhances the need for robust device drivers.

In this article, we suggest enhancing the robustness of device drivers by designing them to be self-stabilizing. Generally, a system is *self-stabilizing* [Dijkstra 1974; Dolev 2000] if it can be started in any possible state, and subsequently converges to a desired behavior. The *state* of a system is an assignment of arbitrary values to the system's variables. Building a system, and specifically device drivers, in this way, ensures that errors will be contained autonomously by each driver, leading eventually to correct behavior of the whole system.

Self-stabilizing operating system (SOS). In order to have a full self-stabilizing system, the other system parts must also be self-stabilizing. This article uses building blocks from our previous work [Dolev and Yagel 2004] where simple self-stabilizing process schedulers are presented, and from Dolev and Yagel [2005], where various memory management schemes are suggested. We also rely on Dolev and Haviv [2006], which addresses self-stabilization of the microprocessor. Thus, based on the idea of fair composition [Dolev 2000], once the microprocessor stabilizes and starts fetching and executing instructions, the system's kernel converges to a legal behavior, in which other programs are executed infinitely often to fulfill the system's goal.

The kernel itself is composed of various layers, each layer providing an easier programming abstraction to the layer above it. When designing a self-stabilizing system, one may use abstraction layers as well, designing the convergence of a certain layer to serve the higher level's convergence [Dolev 2000]. We will now briefly review results from our other works [Dolev and Yagel 2004, 2005], concerning basic hardware resource management. 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 that operating systems must address, are augmented with stabilization requirements. In Dolev and Yagel [2004] we investigated scheduling issues. In Dolev and Yagel [2005] memory management schemes were addressed, and in this article, device drivers are handled.

Scheduling [Dolev and Yagel 2004]. The system is composed of various processes, 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 nonmaskable interrupts mechanisms ensure periodic execution of 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 ongoing execution, and must adapt to different scenarios, for example, starting applications to handle external inputs. Stabilization preservation is achieved by means of monitoring processes and program code restrictions.

Memory management [Dolev and Yagel 2005]. 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.

We suggest 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 processess, and exclusive access is achieved through segmentation and stabilization preservation of the segment partitioning algorithm. Both solutions constrain programs 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 has expired, and returns every such memory portion to the available memory pool for reallocation.

Implementation issues. A demonstration implementation using the Intel Pentium processor architecture [Intel 2007] was composed. This implementation is written in assembly language and is directly assembled into the processor's opcode (we used the NASM open-source assembler [NASM]). The strategy we used for building components for such critical systems was examining, with extra care, every instruction. This is achieved by writing the code directly according to the machine semantics (not relying on current compilers to preserve our requirements), along with line-by-line examination. This style is sometimes tedious, yet it is essential for demonstrating the way to ensure the correctness of a program from any arbitrary initial state. Such a method is especially important when dealing with a component as basic as an operating system kernel. The reader may choose to skip the implementation details. Higher level components and applications can then be composed using methods discussed in Brukman et al. [2003]. We chose the Pentium as a specific example, since it is a widely used and available processor. Similar methods can be applied to other processor architectures. 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. Extensive theoretical research has been done toward self-stabilizing systems [Dijkstra 1974; Dolev 2000; SSS] and recovery-oriented/autonomic-computing/self-repair (e.g., [IBM 2001; Patterson et al. 2002; SUN 2004]). Fault tolerance properties and robustness of operating systems (e.g., [Neumann et al. 1980; Swift et al. 2003]) were also extensively studied. As explained earlier, robustness of device drivers is of great importance in system design. Here, we briefly survey various efforts in this field.

Device driver isolation and monitoring. The micro-kernel system architecture (pioneered in the Mach system [Accetta et al. 1986]) 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 [Tanenbaum and Woddhull 2006], 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 Van Maren [1999] 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 [Barham et al. 2004; LeVasseur and Uhlig 2004; LeVasseur et al. 2004; Swift et al. 2003]. This method relies heavily on the robustness of the core kernel (also known as Virtual Machine Monitor), which we actually address in this article. In Löeser et al. [2004], Leslie and Heiser [2003], and LeVasseur and Uhlig [2004] this is combined with an IO-MMU, which adds hardware protection to I/O access. In Barham et al. [2004] 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 Swift et al. [2004], a monitor that 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.

Language, type safety, and model checking. In Coyotos [Shapiro et al. 2005], the whole kernel, including drivers, is written in a typed language as a stage toward achieving formal correctness. Static analysis of the drivers' code appears in Ball et al. [2006] and Ball and Rajamani [2002]. Vaulet [DeLine and Fähndrich 2001] is a language for describing resources used by protocols that can be enforced by a tool. Recently, those methods were augmented with a general tool for termination checking [Cook et al. 2006] that is used mainly to check device drivers. In Ball and Rajamani [2002], it is claimed that kernels' 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., Chou et al. [2001]) use code analysis to find kernel bugs in general. In Dolev et al. [2005] the requirement for a compiler that preserves self-stabilization of a high-level language program are given.

Singularity is a recent ongoing research project [Hunt et al. 2005] that combines many of the past system research findings in order to achieve better 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 Spear et al. [2006], 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 Hunt et al. [2005], a "malicious driver can program a DMA capable device to overwrite any part of memory").

None of these suggest a design for an operating system or, in particular, a 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). Still, future attempts to supply such drivers can use our concepts and building blocks as a base for obtaining

more sophisticated solutions, possibly using adapted compilers (e.g., DeLine and Fähndrich [2001] and Dolev et al. [2005]).

Organization. In this article, we demonstrate how device drivers can be designed to be self-stabilizing. We start, in Section 2, by demonstrating how the current ATA specification for storage devices (such as hard disks) requires a behavior that can lead a system into undesirable combined states. Based on the definitions and settings presented in Section 3, we demonstrate, in Section 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 5, and an explained implementation that fulfills the self-stabilization requirement appears in the Appendix.

2. A NON-SELF-STABILIZING DRIVER SPECIfiCATION

The AT-Attachment protocol [T13] (standard draft version 8, also historically known as IDE - Integrated Drive Electronics) defines a parallel transport protocol between *host* systems and *devices*. In the following, we will first describe this protocol. Then we will show that the protocol defines interactions that can lead to nonstabilizing 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. Additionally, the device might signal the host through an interrupt line, which we consider as a special register. The host can be configured to be interrupted upon a value change. The required behavior is defined with state diagrams describing states and transitions of both protocol parties. For the purpose of demonstrating the nonstability, 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, return 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 [T13]. 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 [T13]. For simplicity, we combine here these four diagrams into two state machine diagrams, each describing the possible executions of the host and the device, respectively (the reader may choose to examine the original specification, as well). In general, upon a read request, the host checks whether the device is ready (state 1 of Figure 1); it then configures the device, writes the command parameters, and (depending on the value OsInt written by the host) waits for a response through an interrupt (state 2) or by repeatedly checking the status (state 3). The device fills its transfer buffer with part of the requested data (state 1 of Figure 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

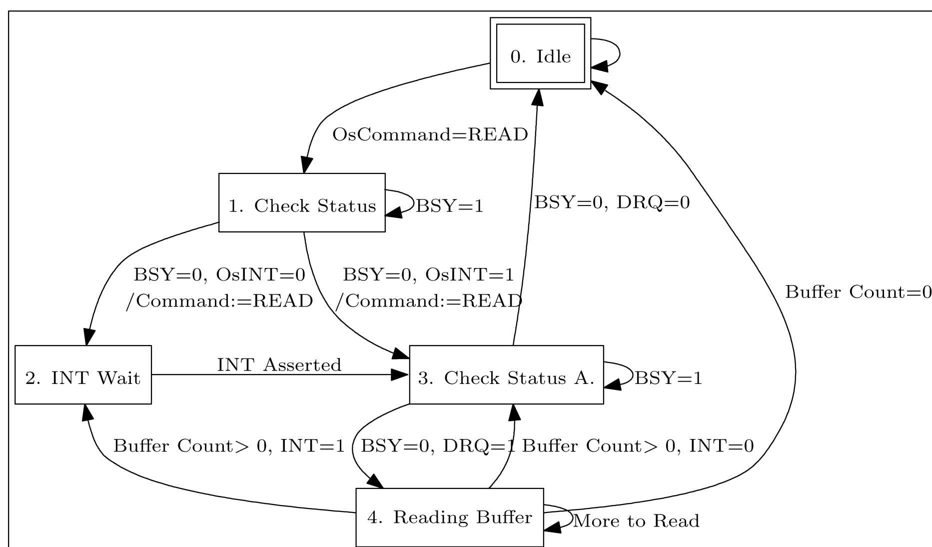

Fig. 1. ATA host state transitions.

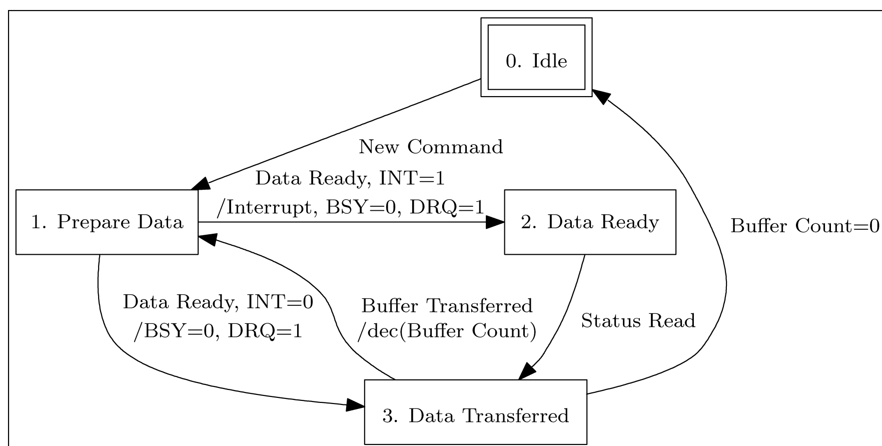

Fig. 2. ATA device state transitions.

state (state 0). For this demonstration, we omit many technical details, that is, selection of devices and media error handling. We show that, in spite of making the model simpler, and also assuming perfect operation of the device mechanics, the execution can still become erroneous.

2.1 Nonstability

The previously described model 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 3. The arrowed path demonstrates one possible correct execution that

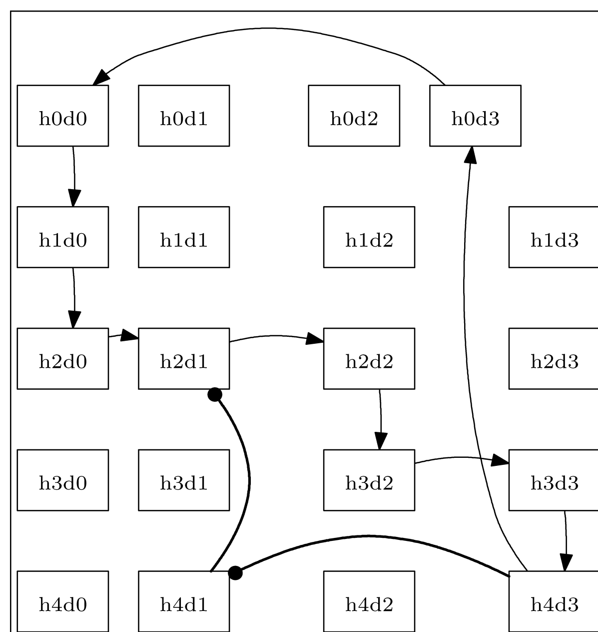

Fig. 3. Union state transitions (h=host, d=device).

is cyclic and includes the combined idle state (marked "h0d0"). We will roughly define here a self-stabilizing systems execution, with regard to drivers and devices, as one which, following any possible state, subsequently converges to a desired behavior. In particular, carrying on its useful task (see Section 3 for more details). Possible nonstabilizing executions are described next:

Deadlock. Due to a transient error, the combined state of the system can change to a state that is not part of the path described previously—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 mean time, 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.

Livelock. Another case is demonstrated in Figure 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," which resides in a different reachable path from the one appearing in Figure 3). It reads in the state register that it is busy, while the device is really nonbusy and actually ready to proceed.

COROLLARY 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.

3. SYSTEM MODEL AND REQUIREMENTS

Settings. We divide the system into four parts: (a) The *operating system*, which contains *processes* (or programs) that can request I/O operations. The operating system contains a special program that schedules all the various processes, including part *b*. (b) The *operating system device driver* (or OS *driver*) is the special program that handles the I/O requests and communicates with the device. (c) The device *controller* is the program executed by a specialized microprocessor (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, for example, 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 1 and 2 are considered transitions made by the OS driver and the device controller, respectively. Concerning the operating system (a), using methods described in our previous works [Dolev and Yagel 2004, 2005], 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, thereby 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 before by the memory manager (see Dolev and Yagel [2005]).

It is also assumed that the I/O device's microprocessor is self-stabilizing, which means that it keeps fetching and executing the device controller program. Methods to achieve such behavior are described in Dolev and Haviv [2006]. 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, for example, bad sectors on the disk media.

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

Definitions. We briefly describe a set of definitions related to states and state transitions (see Dolev and Yagel [2004] and Dolev and Yagel [2005] 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

Table I. ATA Registers

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 regular 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

party is modeled by a program that specifies its behavior. It has a clock that triggers a *step* that 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 of the device controller. An example of such a configuration is "h0d0," which appears in Figure 3, in which both parties are in their idle state. An *execution* is a sequence of alternating configurations and steps $E = (c_1, s_1, c_2, s_2, \ldots)$, such that configuration c_{i+1} is reached from configuration c_i by one step s_i taken by one of the parties. A *request* is a subsequence of an execution, containing steps in which an OS driver writes values to a device's registers. A *reply* is analogously defined in the opposite direction. A configuration such as "h0d0" is called a *safe configuration*, since an execution that starts from this configuration carries out the task of executing I/O commands correctly, that is, according to the ATA specification.

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

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. The result can be a success, for example, data moved according to the command's parameters, or a failure due to bad parameters (causing, e.g., a faulty request or reply), some other transient error (such as dust on the disk surface), or even nontransient 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.

4. 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

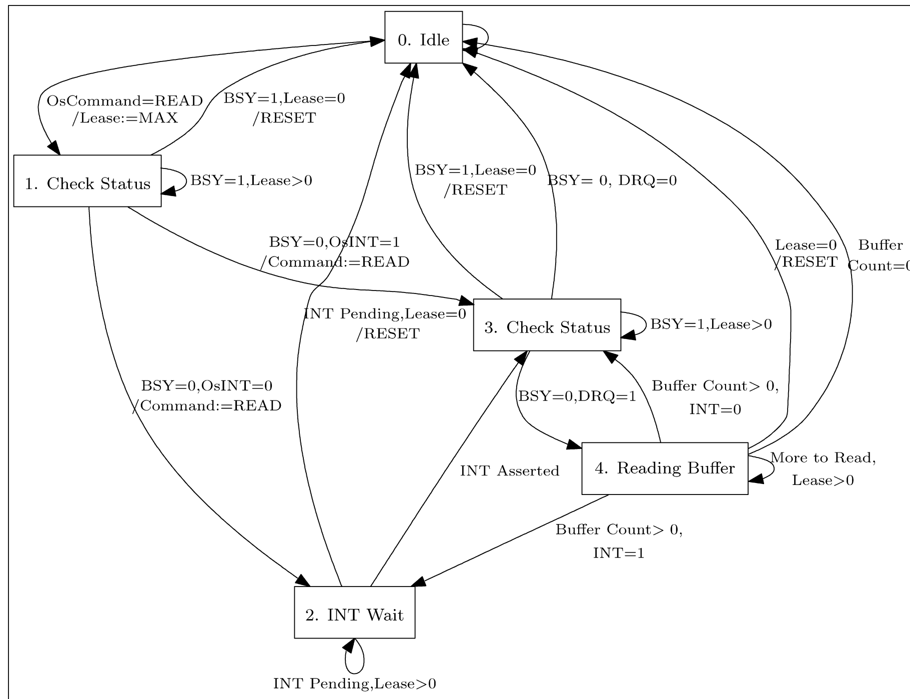

Fig. 4. A leasing host (OS Driver).

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 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.

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 (which is considered unsigned) toward zero, say, in every clock tick. Additionally, the OS driver is augmented with additional transitions guaranteeing 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 4.

Next, we prove the correctness of the leasing solution.

LEMMA 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, that is, a transition back to state 0) 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 toward 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 4.2. *From any configuration a safe configuration is eventually reached.*

PROOF. From Lemma 4.1, the OS driver reaches the idle state infinitely often. 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 4.3. *Since a safe configuration is reached from any state, ping-pong holds.*

LEMMA 4.4. *Eventually progress holds.*

PROOF. From Lemma 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. □

THEOREM 4.5. *A self-stabilizing* OS *driver and device controller combination is achieved.*

PROOF. From Corollary 4.3 and Lemma 4.4, in every infinite system execution, the ping-pong and progress requirements hold infinitely often, so every such execution has a suffix in which both the ping-pong and progress requirements are satisfied, so the OS driver and device controller combination is self-stabilizing. □

Table II. Consistency Check Rules

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

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 nonmaskable interrupt, as described in Dolev and Yagel [2004]. 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 for example, resetting the controller.

For each (abstract) state of the diagrams presented in Figures 1 and 2, there can actually be a set of program counter values that 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 that 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 [Intel 2007], by allocating a full code segment for each PCS (more details appear in Dolev and Yagel [2004]).

The legal device controller PCS and other register values for every such OS driver PCS are included in Table II. 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 4.6. *Eventually, ping-pong holds.*

PROOF. Similarly to Lemma 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 4.7. *Eventually, progress holds.*

PROOF. Similar to the proof of Lemma 4.4. □

COROLLARY 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.*

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 of 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 demonstration implementation of a hard-disk driver is presented. We have tested it using the BOCHS [BOCHS] simulator, and observed that even when arbitrarily changing the contents of the RAM, stabilization is achieved. This implementation can be found in [SOS]. Finally, we predict that provable self-stabilizing operating systems will be an essential part of every critical computing system in the near future.

A. 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 5 through 21. Each code portion is briefly explained, and arguments for its stability are given. This driver implements the lease-based solution of Section 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 4. Note that some "C" language line comments in this source, starting with the ";//" pattern, are added for extra explanation. This is from the source of the ROM-BIOS of Mandrake-Linux, which was adopted by PC emulators such as Bochs [BOCHS].

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 afterwards 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

```
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
```

Fig. 5. Verifying stack memory.

to behave correctly (`ATA 1=>0`). We now go into details of the presented code that forms the stabilizing device driver.

A.1 Verifying Stack Memory Coherence

In lines 1–17 of Figure 5, 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 stack memory within the previously 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.

A.2 Parameter Calculations and Validations

The code in Figure 6 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 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

```
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
```

Fig. 6. Getting base BIOS address and skipping basic checks.

```
;//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
```

Fig. 7. Validate operation code and branch.

can skip some checks, for example, checking that the drive is present and valid, since it is assumed to be enforced by external means.

The code in lines 30–45 of Figure 7 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

```
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
```

Fig. 8. Copying of operation arguments.

```
; 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:
```

Fig. 9. Validate sector count request.

corrupted right after the check is made and before the branch in line 42 (thus causing the processor to jump to unpredicted address), the NMI architecture [Dolev and Yagel 2004], which the system is based on, will eventually recover the system state to a correct state.

In Figure 8, the various operation parameters are just copied inside the stack within the limits mentioned previously.

The code in Figure 9 (lines 72–78) verifies that the number of requested sectors to read/write is legal, that is, between 0 to 127. If this is not the case,

there is a branch to a failure routine (Figure 21), which currently just resets the system. The "C" code returns an error value, which is much less 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 10 (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 can actually be shortened and simplified by comparing to fixed disk addresses).

In Figure 11 (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 toward return from the driver code (`ATA 4=>0`).

A.3 Checking Controller State

The code in Figure 12 (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 13 (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 cannot 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 14 (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 15 (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`).

A.4 Writing Command Parameters and Waiting for Completion

The code in Figure 16 (lines 242–283) writes to the device all command parameters (`ATA 1=>3`). Some parameter calculations that depend on the slaving status can be eliminated if no slaved disk is present.

The code in Figure 17 (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 21)

```
; 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:
```

Fig. 10. Validate request against disk parameters.

```
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
```

Fig. 11. Call to the I/O routine and return status.

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`).

A.5 Read/Write Buffer from Disk

The code in Figure 18 (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 our previous work on memory management [Dolev and Yagel 2005]. A lease is used here again to wait for the disk controller before every subsequent data transfer (`ATA 4=>4`).

```
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
```

Fig. 12. Calculate disk parameters.

```
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.
```

Fig. 13. 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
```

Fig. 14. Reset counters of transferred sectors.

```
;//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
```

Fig. 15. 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
```

Fig. 16. Write command parameters to disk controller (ATA 1=>3).

```
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_:
```

Fig. 17. Wait for disk controller and check controller (`ATA 3=>3, 3=>4, 3=>0`).

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 19, 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 19 (lines 347–378) performs the same steps needed for the output scenario.

The code in Figure 20 (lines 379–394) makes sure that the controller interrupts are enabled for the next I/O command, before executing a return with success indication.

A.6 Failure and Lease Procedures

The code in Figure 21 (lines 395–416) contains the mentioned severe fault handler that raises a general protection error to restart the system, and the code

```
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_
```

Fig. 18. Read transfer loop (ATA 4=>4, 4=>3).

```
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_:
```

Fig. 19. 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
```

Fig. 20. Enable controller interrupts towards return (ATA 4=>0).

```
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
```

Fig. 21. Failure handler and lease procedures.

of the lease procedures. This is followed by the general procedures used to read and write arbitrary memory values, which were omitted here.

REFERENCES

ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. 1986. MACH: a new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference.* USENIX Association, Berkeley, CA, 93–112.

BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. 2006. Thorough static analysis of device drivers. In *Proceedings of European Systems Conference (EuroSys)*. ACM, New York, NY.

BARHAM, P., DRAGOVICH, B., FRASER, K., HAND, S., HO, A., AND PRATT, I. 2004. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for On-Demand IT Infrastructure*.

BRUKMAN, O., DOLEV, S., AND KOLODNER, H. 2003. Self-stabilizing autonomic recoverer for eventual byzantine software. In *Proceedings of IEEE International Conference on Software-Science Technology & Engineering* (SwSTE). IEEE Computer Society, Los Alamitos, CA.

BOCHS. Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net/`

BALL, T. AND RAJAMANI, S. K. 2002. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Terminator: beyond safety. In *Proceedings of the 18th International Conference on Computer-Aided Verification*, Lecture Notes in Computers Science, vol. 3414. Springer, Berlin, Germany. 415–418.

CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER D. 2001. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY.

DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY.

DOLEV, S. AND HAVIV, Y. 2006. Self-stabilizing microprocessor: analyzing and overcoming soft errors. *IEEE Trans. Comput. 55*, 4.

DOLEV, S., HAVIV, Y., AND SAGIV, M. 2005. Self-stabilization preserving compiler. In *Proceedings of the 7th International Symposium on Self-Stabilizing Systems* (SSS). Lecture Notes in Computer Science, vol. 3764. Springer, Berlin, Germany.

DIJKSTRA, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Comm. ACM, 17*, 11, 643–644.

DOLEV, S. 2000. *Self-Stabilization*, The MIT Press, Cambridge.

DOLEV, S. AND YAGEL, R. 2004. Toward self-stabilizing operating systems. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems* (SAACS,DEXA), 684–688,

DOLEV, S. AND YAGEL, R. 2005. Memory management for self-stabilizing operating systems. In *Proceedings of the 7th Symposium on Self Stabilizing Systems (SSS)*. Lecture Notes in Computers Science, vol. 3764. Springer, Berlin, Germany.

HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. 2005. An overview of the Singularity project, Tech. rep. MSR-TR-2005-135, Microsoft Corporation, Redmond, WA.

LÖESER, H. J., MEHNERT, F., REUTHER, L., POHLACK, M., AND WARG, A. 2004. An I/O architecture for mikrokernel-based operating systems. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Washington, DC.

IBM. 2001. Autonomic computing initiative. `http://www.research.ibm.com/autonomic`.

INTEL CORPORATION. 2007. *The IA-32 Intel architecture software developer's manual*. `http://developer.intel.com/design/pentium4/documentation.htm`.

LESLIE, B. AND B. HEISER, B. 2003. Towards untrusted device drivers. Tech. rep. UNSW-CSE-TR-0303, School of Computer Science and Engineering UNSW.

LeVasseur, J. and Uhlig, V. 2004. A sledgehammer approach to reuse of legacy device drivers. In *Proceedings of the 11th ACM SIGOPS European Workshop.* ACM, New York, NY.

LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. 2004. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Washington, DC.

Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Washington, DC.

nasm. The netwide assembler. http://nasm.sourceforge.net.

Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L. 1980. A provably secure operating system: the system, its applications, and proofs, Tech. rep. CSL-116, SRI International.

Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP).* ACM, New York, NY.

Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. 2002. Recovery oriented computing (ROC): motivation, definition, techniques and case studies. Tech. rep. UCB/CSD-02-1175, UC Berkeley Computer Science, Berkeley, CA.

Swift, M. 2005. Improving the reliability of commodity operating systems, Ph.D. dissertation, University of Washington.

Swift, M., Annamalai, M., Bershad, B. N., and Levy, H. M. 2004. Recovering device drivers. In *Proceedings of the 6th ACM / USENIX Symposium on Operating Systems Design and Implementation (ODSI).* USENIX Association, Washington, DC.

Swift, M., Bershad, B. N., and Levy, H. M. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP).* ACM, New York, NY.

Shapiro, J., Doerrie, M. S., Northup, E., Sridhar, S., and Miller, M. 2005. Towards a verified, general-purpose operating system kernel. http://www.coyotos.org.

sos. http://www.cs.bgu.ac.il/~yagel/sos.

Spear, M., Roeder, T., Hodson, O., Hunt, G., and Levi, S. 2006. Solving the starting problem: device drivers as self-describing artifacts. In *Proceedings of the EuroSys.* ACM, New York, NY.

sss. http://www.selfstabilization.org.

Sun Microsystems, Inc. 2004. Predictive self-healing in the Solaris™ 10 operating system. White paper http://www.sun.com/software/solaris/ds /self_healing.pdf.

T13. International Committee for Information Technology Standards. ATA Storage Interface - T13/1532D Vol. 2. Rev. 4a (working drafts). http://www.t13.org/#Projects.

Tanenbaum, A. S. and Woodhull, A. S. 2006. *Operating Systems Design and Implementation*. 3rd Ed, Prentice Hall.

Van Maren, K. T. 1999. The Fluke device driver framework. Master's thesis, The University of Utah.

Received February 2007; revised July 2008; accepted September 2008