

Toward Self-Stabilizing Operating Systems*

(Extended Abstract)

Shlomi Dolev and Reuven Yagel[†]

Department of Computer Science

Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel

{dolev,yagel}@cs.bgu.ac.il

Abstract

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. The second, 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. The last approach presents an example of a tailored self-stabilizing very-tiny operating system. Prototypes using the Intel Pentium processor were composed.

1 Introduction

The robustness of an operating system is, in some cases, more important than its performance [11]. 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 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.

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 [18] 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. In this work we propose several approaches for designing automatic recovering operating system that are based on the well defined and well understood self-stabilization paradigm [4, 5]. 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 systems 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 [7] 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 [8] is used here to show that once the underlying microprocessor stabilizes 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. In this work we consider the important layer of the operating system. Operating systems are essential parts of most computer systems. The operating system manages the hardware resources, and forms an abstract (virtual) machine that is convenient to program by higher level applications developers.

Operating systems are not self-stabilizing: The operating system is the largest software constantly executed by a processor. Fault free software is a hard task to achieve (see, e.g., [2]). When the operating system is designed for a specialized restricted task, such as the TinyOS [14], formal methods of verification may assist in achieving fault free software. Still, the resulting system may fail due to a tran-

*Partially supported by Microsoft, MFAT, IBM, NSF, STRIMM, Rita Altura Trust Chair in Computer Sciences and Lynn and William Frankel Center for Computer Sciences.

[†]Also with Rafael WTEC, Mizpe-Ramon, Israel.

sient fault (e.g., a soft-error). Therefore a self-stabilizing approach is a must in such basic and on-going components such 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. 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 [13]. 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 [20]. Also when in a system management mode (say, due to transient faults) the NMIs are disabled [13](Vol. 3, Sec. 13.7), so the processor has masked NMI states.

The self-stabilizing approach for modeling faults is orthogonal to the Byzantine faults model [16], both approaches can in fact be combined [9]. While in the Byzantine model faulty processors 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 processors in the system execute a correct code (e.g., [9]). The requirement for code correctness (of at least two thirds, or so, of the processes) is obvious, even in the case of Byzantine faults [16]. If all the processors 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 the 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 an infinite loop. The processor has an operation to set this register value. A similar scenario is possible when the interrupt table itself is corrupted.

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 call this approach the black-box based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We call this approach the tailored solution approach. We present three design solutions in the scale of the black-box to the tailored solutions. The first simplest technique we propose 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. Then we present a tailored very tiny self-stabilizing design for components of an operating system.

Previous work: Extensive theoretical research has been done toward self-stabilizing systems [4] and recovery-oriented/autonomic-computing/self-repair, see [18, 3, 2, 7, 11] and the references therein. Some systems are built to cope with severe fault combinations such as [3, 6]. Some operating systems, including Microsoft Windows XP [17] and EROS [15] use checkpointing to gain fault-tolerance. Other systems [1, 19] supply monitoring layer for ubiquitous operating systems like Linux and Windows (these systems use the mentioned NMI). However, none of the above suggest a design for an operating system that can withstand any combination of transient-faults.

Paper organization: The rest of the paper is organized as follows. Section 2 describes our model for the operating system and its interaction with the processor and the (users) applications. Section 3 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 adds details for reinstall of the executable portion, monitoring the state of the operating system and assigning a legitimate state whenever required. Lastly, a tailored, very tiny, self-stabilizing design for components of an operating system is presented. Details and proofs are omitted from this extended abstract (see [10] for an extended version).

2 System Settings

In this section we define 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 model the *system* by a tuple $\langle \text{processor}, \text{memory}, I/O \text{ connectors} \rangle$. 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 [13]{2/3.2}, 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 [7].

A *system configuration* is a processor state and the content of the system memory. A *system execution* $E = (c_1, a_1, c_2, a_2, \dots)$ 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 *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 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 [12], 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: An additional watchdog device 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 its execution from where it stopped.

Implementation issues: Right after a computer is booted, the core of an operating system is loaded to the computers Random Access Memory (RAM) which will we simply call *memory*. This is done by procedures residing in nonvolatile Read-Only-Memory (ROM) such as EPROM (Erasable Programmable ROM) which is also called Basic-Input-Output-System (BIOS). 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 like procedure which will reside in ROM. This procedure will periodically reinstall the operating system from a (CD) ROM image and then restart the execution of the oper-

ating system.

4 Reinstall Executable and Monitor State

In this section we present approaches that couple the reinstall/repair with 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 used in Section 3 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 a monitor/restarter (see [2]) 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.

5 Tailored Very Tiny Self-Stabilizing Operating System

Alternatively 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.

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. There are several possible requirements that the scheduler should satisfy. Here we require:

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.

Stabilization preserving: In addition, we would like to ensure that the scheduler preserves the self-stabilization prop-

erty 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.

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 `jbε`) 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. Similarly, we assume that the references to RAM addresses are fixed in the code (rather than indirect addresses). To simplify our claims we also require that the code of the processes does not contain loops.

The above assumptions are used to prove that eventually every self-stabilizing process eventually stabilizes.

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 another 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 pre-defined 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 (details follow).

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

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`, `jbε`) 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 code does 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 [13]{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 nops up to a fixed length (an assembler post processing might be relevant here) and additional check by the scheduler that the `ip` value holds an address of an instruction start, before the execution of `iret`. (Section 3 did not have to address this issue, since the NMI set 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 value of the general purpose registers, flag register and the value of the program counter. The code of each process will be (repeatedly) read by the scheduler from 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 used for Section 3, alternately, one may assume that a special process among the N processes, that reside 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 3. When 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.

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). See [10] for more details, code and proofs.

References

- [1] Francois Armand. "ChorusOS Features and Architecture overview", Sun Technical Report, December 2001.
- [2] 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.
- [3] 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.
- [4] S. Dolev. *Self-Stabilization*, The MIT Press, Cambridge, 2000.
- [5] E. W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control", *Communications of the ACM*, Vol. 17, No. 11, pp. 643-644, 1974.
- [6] 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.
- [7] S. Dolev, Y. Haviv. "Self-Stabilizing Microprocessor, Analyzing and Overcoming Soft-Errors", *17th International Conference on Architecture of Computing Systems (ARCS04)*, pp. 31-46, 2004.
- [8] S. Dolev, S. Moran, A. Israeli. "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, pp. 103-117, 1990.
- [9] 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.
- [10] S. Dolev and R. Yagel. "Toward Self-Stabilizing Operating Systems". Technical report, #04-01, Computer Science, Ben-Gurion University, Beer-Sheva, Israel, March 2004.
- [11] A. Fox, D. Patterson. "Self-Repairing Computers", *Scientific American*, June, 2003
- [12] 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.
- [13] Intel. "The IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. Volume 2: Manual Reference. Volume 3: System Programming guide", <http://developer.intel.com/design/pentium4/manuals/>, 2003.
- [14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. "System Architecture Directions for Networked Sensors", ASPLOS 2000.
- [15] 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.
- [16] 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.
- [17] Microsoft. "Windows XP/Office XP Feature Overview". <http://www.microsoft.com/windowsxp/pro/evaluation/overviews/windowsxpofficexp.asp>.
- [18] P. C. Murley, G. R. Srinivasan. "Soft-error Monte Carlo modeling program, SEMM", IBM Journal of Research and Development, volume 40, Number 1 pages 109-118, 1996.
- [19] 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/>
- [20] <http://users.win.be/W0005997/GI/nmi.html>