Toward a Self-Stabilizing Operating System

Shlomi Dolev∗  Reuven Yagel†

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel

March 8, 2003

Abstract

Every paper starts with an abstract which should be short and written in a very special style. The abstract is used mostly by people who do not wish to read the paper, and are looking for short description of what’s in it. The abstract is written last. In the first papers your write, it is usually better to let your advisor write it for you.

1 Introduction

For a system to be self-stabilizing, it is necessary and sufficient that the various algorithms and components be composite as described by [Dolev 2000]. In a real computing system, the hardware has to have this property, as well as the various software components up to the algorithms which determine or fulfill the system behavior. One such essential part of most computer systems is an Operating-System. Its main targets are managing the hardware resources and presenting the higher level software, an abstract (virtual) machine [Tanenbaum 2001]. This article describes one step of the research toward a Self-Stabilizing Operating System - SSOS.

In order to supply an SSOS, we could take the top-down approach and start developing self-stabilizing versions of various standard OS algorithms, like process scheduling or memory management. We could even apply those changes to some open-source OS. This direction, while very practical, is very difficult to prove because the system built is already composed of various quite complicated components.

∗dolev@cs.bgu.ac.il
†yagel@cs.bgu.ac.il
Instead we took a bottom-up approach. Therefore we looked for the most elementary part of an OS and try to stabilize it. Usually right after a computer is booted, the core of an OS is loaded to the computer's Read Access Memory (RAM) which will be simply referred to as 'memory'. This is done by procedures residing in nonvolatile Read-Only-Memory (ROM) such as EPROM (Erasable Programmable ROM), usually referred as Basic-Input-Output-System (BIOS). After the OS is loaded to memory, as a minimum requirement we would like to guarantee that the code that was loaded is correct. Assuming that the OS code itself is self-stabilizing, it might happen that the memory holding the OS code will be corrupted (e.g. by soft errors), leading to a situation that the OS does not converge to a valid state. In order to deal with this problem we would like our OS to have an essential procedure which will reside in ROM, and will periodically load the OS from a ROM image and start it again. This part of the OS will be called the OS Stabilizer.

The Intel's Pentium 4 Processor was chosen for implementing the SSOS. In the next section we will sketch the algorithm and prove its correctness. Then we will describe the platform for implementing the solution and prove that the solution is correct for this platform. Afterwards we will say a few words about simulation of the solution and future work.

2 The OS Stabilizer

Assumptions We will lay out a few general assumptions about the system. We have an underlying stabilized hardware which especially contains a CPU that keeps fetching, decoding & executing instructions from memory. This processor uses an address space which is combined of ROM and RAM parts. Only the content of the ROM is guaranteed not to be changed by any means. The RAM content might be changed by the processor, and also by other means; especially various errors.

Additionally we have a mechanism which from time to time is guaranteed to make the CPU stop the regular cycle and "jump" to a procedure residing in ROM and start executing instructions from there.

Let us summarize the assumptions: We assume the underlying CPU is self-stabilizing. For our purpose it means:

- a) There is a copy of the OS which resides in ROM without changes, and accessible to the processor.
- b) The processor always continues to fetch instructions and execute them. Each fetch-decode-execute cycle is atomic.
- c) Between every cycle, it can be interrupted unconditionally from outside. (Except when handling this interrupt) This interrupt causes it to jump unconditionally to the stabilizer code.
d) There is a mechanism that does interrupt it once in a while, sometimes this mechanism is called a 'watchdog'.

Algorithm  
Upon entering the stabilizer code:

1. Setup the processors state for enabling the correct execution of the stabilizer.
2. Copy OS code from ROM image to RAM.
3. Enable external interrupts and Jump to OS code.

Proof: From assumptions (b) (c) and (d) we know that once in a while the OS loader code will be executed and completed. (a) guarantees that we have the original code image to copy from. From (b) we know that steps 1-4 will be executed and come to a conclusion. From step 1 we know that the processor state is valid and ready for the copy process. From step 2 we know that the OS original code is again residing in RAM ready for execution. From step 3 we know that, in one atomic step, the processor returns to a state which enables the next stabilizing stage and starts executing the OS again.

This loading procedure assures that: once in a predefined period of time, which is not dependent on any memory resident software, the OS starts all over again with the original code, thus enters a valid state.

3 Implementation

Hardware Platform  
In this section we will describe the hardware platform that we chose to implement the stabilizing OS. We will concentrate on the needed information for describing the solution.

We chose the Intel’s Pentium 4 Processor for implementing the SSOS. This is a 32 bit, ubiquitous processor which we already had in hand. Additionally it has published & available manuals which we used to write the code and verify its correctness. This processor is built from dozens of millions transistors and is also a commercial one so we can’t guarantee that we know it’s full state, but we will describe its state and environment according to the known registers it has. The main source for information about the processor is Intel’s manuals [Intel 2003]. When referring to these manuals we will sometimes only cite the information source, since it is not in the scope of this article to go into so much detail. The reference will be in the form {volume/chapter.section}.

Real-address mode {3/16.1}  
The IA-32 architecture processor family, including the Pentium processor, has 3 (or 4) operation modes: Protected mode, Real-address mode, Virtual-8086 mode (which is part of the protected...
mode) and System Management mode. In the 'Real-address Mode' the processor executes programs written for the rather old Intel 8086 processor. Programs running in this mode can explicitly use processor resources added in later generations. There is a 2nd mode called Virtual-8086 mode that is part of the regular mode (protected mode) which enables concurrent running of real-mode programs. When the processor is booted up it is put in the real-mode; probably for backward compatibility reasons. We chose to work in the real mode because of its relevant simplicity and the availability of documentation. Entering real mode is done by clearing bit 0 of the CR0 control register. Other flags in this register are relevant to protected mode and controlling memory caching inside the processor. The process of entering the real mode is detailed in 3/9.9.2.

**Processor environment & state** We will now describe the environment in which the processor operates in the real-address mode.

**Address Space** The process supports addressing of 1-MByte of physical address space. This space is divided into segments of 64Kbytes. The base of a segment is specified by a 16 bit selector (stored in one of the segment registers) which is extended with an additional 0 to form 20bit offset from address 0 in the address space. To address a specific address, an operand of 16 bit is added to the segment offset to form the actual address.

Stack - one stack, 16-bit wide. Resides in memory.

Interrupt Vector Table - Single, called 'the interrupt vector table' or just 'interrupt table'. It is an array of 4-byte entries, each 4 bytes is an address which is called a pointer (or vector) to a procedure in memory that service the interrupt. The base of the table is stored in the IDTR register, and can be changed using the 'lidt' instruction. In contrast to this architecture we must assume that this table can not be relocated and its content, especially the vector for the OS stabilizer, is fixed, i.e. it is in ROM. This assumption, while not implemented in standard computers, is not so unreal since occasionally there are suggestions to implement full OSs on ROM.

**Interrupts {3/16.1.4}** As said, we need a mechanism for starting the stabilizer procedure once in a while. The natural candidate is the interrupt mechanism built into the processor. When an (internal or external) interrupt occurs the processor stops executing the current program and does the following: Pushes the value of the FLAG register onto the stack. (Remark: From {3/16.1.4} it might appear that this step follows the next step. This is not true). Pushes the values of CS register (segment address of Program Counter) & IP register (The lower 16 bit of the 32-bit Program Counter) registers onto the stack. Clears the IF flag in the FLAG register. Transfers
program control (i.e. loads CS & IP) with the location specified in the interrupt table. After servicing the interrupt, the IRET instruction is usually called, reversing the steps to resume the interrupted program.

NMI - Non-maskable Interrupt is generated by external hardware that asserts the NMI pin. (Starting from Pentium 4 it can also be generated internally by instructions). When this happens the processor immediately moves to the handler pointed to by vector 2. This interrupt cannot be masked by the IF flag, so it is guaranteed to make the processor react. This happens even if the instruction 'hlt' was called. {2/3.2 hlt} While an NMI is handled, additional NMIs are disabled, until the 'iret' instruction. 3/18.2.2 NMIs can be disabled with external circuitry. {3/9.9.1}

Watchdog - A special hardware unit outside of the processor, that is guaranteed to assert one of the processor’s legs once in a predefined period of time. Often, it is connected to the 'RESET' leg in order to watch if some important process is stuck, and then restart the machine. However, it can be connected to the NMI pin, thus insuring a call to a critical procedure ad the OS stabilizer. [Castro and Liskov 2000] assumes this mechanism also for a fault-tolerant system.

Timer - The processor has a built in timer that ticks every 55 milliseconds. Most interrupts including the timer interrupt can be discarded by the CPU as a result of executed instructions that are handled by the CPU, and change its state. However we used this interrupt in our solution simulation, to simulate the NMI interrupt.

**Instructions** In real mode the processor has a set of instructions that can be carried as specified by Intels Manual. {3/16.1.3} We assume that only these instructions together with the external interrupts are causing state changes to the processor.

**Registers** The CPU state is assumed to be totally determined by the values of its different registers, so we next list all of the published Pentium registers and there function. This is based on {1/3} and {3/16.1}.

There are - 8 general purpose registers, 6 segment registers, EFLAG register (Process State Word) and EIP register (Extended Program Counter). These registers are used, among other tasks, for program flow control and addressing memory (including the stack) which will be used by our code. Since we are operating in the real-address mode, only the low 16-bits of these registers are relevant to the processor state.

Control registers - 5 control registers, determine the operation mode of the processor. More details in {3/2}.

We will mention here briefly the other registers, which are not assumed to influence our code. x87 FPU registers - floating point operations. MMX and XMM registers - SIMD operations. Memory management registers - 4
OS_SEGMENT equ 0x1000
OS_ROM_SEGMENT equ 0x3000

; enter realmode
1. mov cr0, 0

; copy OS image
2. mov ax, OS_ROM_SEGMENT
3. mov ds, ax
4. mov ax, OS_SEGMENT
5. mov es, ax
6. mov si, 0x00
7. mov di, 0x00
8. mov cx, 0x100
9. cld
10. rep movsb

; prepare for journey back
11. mov ax, cs
12. mov ss, ax
13. mov sp, 0xFFFF
14. pushf
15. push word OS_SEGMENT
16. push word 0x0

; enable NMI & jump to OS code.
17. iret

registers for specifying the location of data in protected mode, which we do not use.

Also part of the basic execution environment are the IOports, which will not be used either. There are also performance-monitoring counters and internal caches and buffers. {3/2.1.7}

The OS stabilizer procedure  The loading procedure will be presented in simple assembly language, which is directly assembled into the processor’s opcode. It was assembled with NASM an open-source assembler [NASM].

The loading procedure will be presented in simple assembly language, which is directly assembled into the processor’s opcode. It was assembled with NASM an open-source assembler [NASM].
Proof 1) The CPU can always be interrupted by an NMI. 2) We know that the CPU will get to the stabilizer code because of the external watchdog. 3) We assume the OS stabilizer algorithm is correct and has a finite number of instructions to perform. 4) We assume that the processor is not halting and will not be interrupted during this execution, since it is executing an NMI procedure. Thus the stabilizer code will be executed to the last instruction. 5) The code is residing in ROM, so it is guaranteed not to be corrupted. 6) The code does not use any variable, so it is not dependent on memory values. 7) The code does not count on any previous processor state. Scanning the code for register names shows that all of them appear for the first time as a left operand of the 'mov' instruction, thus assigning them a new value. The only exception for this assertion is the CS register, whose value is assigned by the processor when the NMI is triggered to the loader code segment. Otherwise we wouldn’t be able to execute the stabilizer code at all. 8) The copy to memory uses operations whose meaning is completely determined by: Being in real mode - bit 0 of CR0 register is set in line 1. The values of registers DS, ES, SI, DI and CX which were set in lines 2-8. The direction flag which is set in line 9. This is in accordance with the instruction reference at \{2/3.2\} 9) We jump to the OS code with the 'iret' instruction which enables again NMIs and takes the new PC values from the stack that is set in lines 11-16. Therefore after executing the stabilizer code we have a fresh copy of the OS code in memory, the processor state is a valid one and ready to execute the OS. We also know that this process will start again in the future.

Simulation description Loader in RAM with variable Timer interrupt OS code is Memory layout: 0 Interrupt vector table 1000 2nd stage loader 7C00 1st stage loader 10000 OS A0000 Watchdog handler - ROM A0100 OS Image - ROM

4 Future Work

Warm Boot.
Full OS that checks it is in a stable situation.

References


More