Memory Management for Self-stabilizing Operating Systems*
(Extended Abstract)

Shlomi Dolev and Reuven Yagel

1 Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{dolev, yagel}@cs.bgu.ac.il
2 Rafael 3M, POB 2205, Haifa, Israel

Abstract. This work presents several approaches for designing the memory management component of self-stabilizing operating systems. We state the requirements which a memory manager should satisfy. One requirement is eventual memory hierarchy consistency among different copies of data residing in different (level of) memory devices e.g., RAM and disk. Another requirement is stabilization preserving where the memory manager ensures that every process that is proven to stabilize independently, also stabilizes under the (self-stabilizing scheduler and the) memory manager operation. Three memory managers that satisfy the above requirements are presented. The first allocates the entire physical memory to a single process in every given point of time, the second uses fixed partition of the memory among processes, and the last uses memory leases for dynamic memory allocations.

1 Introduction

This work presents new directions for building self-stabilizing memory management as a component of a self-stabilizing operating system kernel. A system is self-stabilizing [7,8] 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. The usefulness of such a system in critical and remote systems cannot be over estimated. Entire years of work maybe lost when the operating system of an expensive complicated device e.g., a spaceship, may reach an arbitrary state due to say, soft errors (e.g., [14]), and be lost forever.

An operating system kernel usually contains the basic mechanisms for managing the hardware resources. The classical Von-Neumann machine includes a processor, a memory device and external $i/o$ devices. In this architecture memory management is an important task of the kernel of the operating system. Our memory management uses the primitive building blocks from [10] where simple self-stabilizing process schedulers are presented.

* Partially supported by Rafael, Microsoft, IBM, NSF, Intel, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences and Lynn and William Frankel Center for Computer Sciences.
Management of memory influenced the development of computer architecture and operating systems [2]. Various memory organization schemes and appropriate requirements have been suggested during the years. Here we add two important requirements called the eventual memory hierarchy consistency requirement and the stabilization preserving requirement. Since memory hierarchies and caching are key ideas in memory management, the memory manager must eventually provide consistency of the various memory levels. Secondly, once stabilization is proven for a process, the fact that process and scope switching occurs and memory is actually shared with other processes, will not damage the stabilization property of the process. These requirements are an addition to the usual efficiency concerns which operating systems must address.

We present three basic design solutions that, roughly speaking, follow the evolution of memory management techniques. The first approach allocates the whole available memory to the running process, thus ensuring exclusion of memory access. This method is simple but not efficient, since each process switch requires expensive disk operations. The second solution partitions the memory among several running processes, exclusive access is achieved through segmentation and stabilization of the segment partitioning algorithm. Both solutions constrain program referencing to addresses in the physical memory only (or even in the partition size) and allow only static use of memory. Then we present lease based dynamic schemes, where the application must renew memory leases in order to ensure the correct operation of a self-stabilizing garbage collector.

Demonstration implementations (which appear in [11]) using the Intel Pentium processor architecture [13] were composed. The implementations are written in assembly language, and are directly assembled into the processor’s opcode (in our experiments we have used the NASM open-source assembler [16]). The methodology we used for building such critical systems is to examine, 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), together with line by line examination. This style is sometimes tedious, but is essential to demonstrate the way one should ensure the correctness of a program from any arbitrary initial state. Such a method is specially important when dealing with such a basic component as an operating system kernel. Higher level components and applications can then be composed in ways discussed in [4].

The Intel Pentium processor contains various mechanisms which support robust design of memory management like segmentation, paging and ring protection. However, the complexity of the processor (partially due to previous processors compatibility requirements) carries a risk of the entering into undesirable states, thereby causing undesirable execution. Our proof and prototype show that it is possible to design a self-stabilizing memory manager that preserves the stabilization of the running processes which is an important building block of an infrastructure for industrial self-stabilizing systems.

Previous Work: Extensive theoretical research has been done toward self-stabilizing systems [7,8,21] and recovery-oriented/autonomic-computing/self-repair, e.g., [12,17,22]. Fault tolerance properties of operating systems (e.g., [19],