

Memory Management for Self-Stabilizing Operating Systems*

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 the memory management component of self-stabilizing operating systems. We state the requirements 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 preservation* a condition in which the memory manager ensures that every process that is proven to stabilize independently, stabilizes under the (self-stabilizing scheduler and) memory manager operation too. Three memory managers that satisfy the above requirements are presented. The first allocates the entire physical memory to a single process at every given point in time. The second one uses fixed partition of memory between processes, while the last one uses memory leases for dynamic memory allocations. The use of leases in the scope of memory allocation in the operating system level is a new and important aspect of our self-stabilizing memory management.

*Partially supported by Rafael, IBM, NSF, Intel, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences and Lynn and William Frankel Center for Computer Sciences.

[†]Also at: Rafael 3M, POB 2205, Haifa, Israel.

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* [8, 9] if it can be started in any possible state and subsequently it converges to a desired behavior. A *state* of a system is an assignment of arbitrary values to the system's variables. The necessity of such a system in critical and remote systems cannot be over estimated. Entire years of work can be wasted if the operating system of an expensive complicated device e.g., a spaceship, reaches an arbitrary state due to say, soft errors (e.g., [15]), and is lost forever.

An operating system kernel usually contains the basic mechanisms for managing 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 operating system's kernel. Our memory management uses the primitive building blocks from [11] where simple self-stabilizing process schedulers are presented. We also rely on [7], which addresses self-stabilization of the microprocessor. Thus, based on the idea of fair composition [9], once the microprocessor stabilizes and starts fetching and executing instructions, the scheduler converges to a legal behavior, in which other programs are executed infinitely often.

Memory management has influenced the development of computer architecture and operating systems [3]. Various memory organization schemes and appropriate requirements have been suggested throughout the years. In this work, we are adding two important requirements, named the *eventual memory hierarchy consistency* requirement and the *stabilization preservation* requirement. Since memory hierarchies and caching are key ideas in memory management, the memory manager must eventually provide consistency of the various memory levels. Additionally, once stabilization for a process is established, the fact that process and scope switching occurs and that memory is actually shared with other processes, will not damage the stabilization property of the process. These requirements are an addition to the usual efficiency concerns which operating systems must address. Usually, memory management in operating systems is handled with the assistance of quite a complex state, for example page tables. A minor fault in such a state can lead to writing wrong data onto the disk (violation of consistency) or even to corruption of some process' state (violation of preservation). Unless the system is self-stabilizing, the corruption of the tables may never be corrected and the tables' internal consistency may never be re-established. We present three basic design solutions that, roughly speaking, follow the evolution of memory management techniques. The first approach allocates the 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 between several running processes and exclusive access is achieved through segmentation and stabilization of the segment partitioning algorithm. Both solutions constrain program referencing to addresses in the physical memory only (or even in the partition size) and allow only static use of memory. Following this, we present lease based dynamic schemes, in which the application must renew memory leases in order to ensure the correct operation of a self-stabilizing garbage collector.

Demonstration implementations (which appear in the appendices) using the Intel Pentium processor architecture [14] 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 [19]). The strategy we used for building 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 is essential for demonstrating the way one should 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. Higher level components and applications can then be composed using ways discussed in [1]. The reader may choose to skip the implementation details. The Intel Pentium processor contains various mechanisms which support the robust design of memory management like segmentation, paging and ring protection. However, the complexity of the processor (partially due to previous processors' compatibility requirements) carries a risk of entering into undesirable states, and thereby causing undesirable execution. Our proof and prototype show that it is possible to design a self-stabilizing memory manager that preserves the stabilization of the running processes which is an important

building block of an infrastructure for industrial self-stabilizing systems.

Previous work: Extensive theoretical research has been done towards self-stabilizing systems [8, 9, 24] and recovery-oriented/autonomic-computing/self-repair, e.g., [13, 20, 25]. Fault tolerance properties of operating systems (e.g., [22]), including the memory management layer, were extensively studied as well. For example, in [4], important operating system memory regions are copied into a special area for fast recovery. The design of the Multics operating system pioneered issues of data protection and sharing, see [6] and [21]. The algorithms presented here enforce consistency of the data structures used for memory management. In order to use more complex data structures, the work of [12] is relevant for achieving general stabilization of data structures. [10] addresses the issue of automatic detection and error correction in common high-level operating system data structures (although, not in a self-stabilizing way). Leases are commonly used as a general fault-tolerant locking mechanism (see [16]). In [18], leases are used to automatically remove subscriptions in publish-subscribe systems. However, none of the above suggest a design for an operating system, or, in particular, memory management that can automatically recover from an arbitrary state (that may be reached due to a combination of unexpected faults).

Paper organization: In the next section we define the system settings and requirements. The three solutions: total swapping, fixed partition and dynamic memory allocation, are presented in Section 3, Section 4 and Section 5, respectively. Concluding remarks appear in Section 6 and code snippets with explanations appear in the Appendix.

2 System Settings, Assumptions and Requirements

We start with a brief set of definitions related to states and state transitions (see [7, 11] for more details). The *system* is modeled by a tuple $\langle \text{processor}, \text{memory}, \text{i/o connectors} \rangle$. The *processor* (or microprocessor) is defined by an operation manual, e.g., Pentium [14]. The *processor state* is defined by the contents of its internal memory (registers).

The *registers* includes a *program counter* (*pc*) register and a *processor status word* (*psw*) register, while the latter determines the current mode of operation. In particular, the *psw* contains a bit indicating whether interrupts are enabled.

A *clock tick* triggers the microprocessor to *execute a processor step* $ps_j = (s, i, s', o)$, where

the inputs i and the current state of the processor s are used for defining the next processor state s' and the outputs o . The *inputs and outputs* of the processor are the values of its *i/o connectors* whenever a clock tick occurs. The processor uses the *i/o connectors* values for communicating with other devices, mainly with the memory, via its data lines. In fact, the processor can be viewed as a transition function defined by e.g., [14]. A *processor execution* $PE = ps_1, ps_2, \dots$ is a sequence of processor steps such that for every two successive steps in PE , $ps_j = (s, i, s', o)$ and $ps_{j+1} = (\bar{s}, \bar{i}, \bar{s}', \bar{o})$ it holds that $s' = \bar{s}$.

The *interrupt connector* which is connected to external i/o devices, is used for signaling the processor about (urgent) service requests. The NMI (Non-Maskable Interrupt) *connector* role is similar to the interrupt connector, except that the NMI request is not masked by the interrupt flag. In the Pentium, whenever one NMI is handled, other NMIs are ignored until an `iret` operation is executed.

The *memory* is composed of various devices (Figure 1 presents some common *memory hierarchy*). Here we consider *main memory* and *secondary storage*. The main memory is composed of ROM and RAM components. Read-only parts are assumed non-volatile. The secondary storage is also organized as a combination of read-only

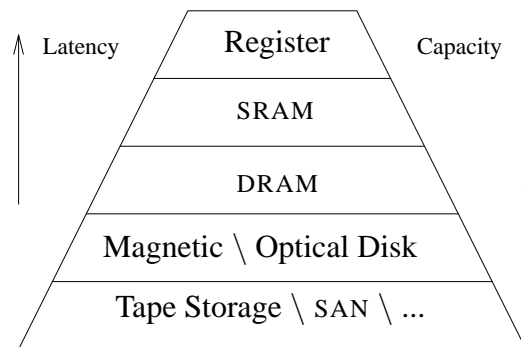


Figure 1: A Common Memory Hierarchy

parts, such as CD-ROM and other disks. The read-only requirement is compulsory for ensuring correctness of the code. Otherwise, the Byzantine fault model [17] must be assumed. Processor caches, at least in the current Pentium design can not be controlled directly by the operating system, and are not, therefore, considered here.

The *i/o state* is the value of the *connectors* connecting to peripheral devices. We assume that any information stored in the interface cards for these devices is also part of the memory.

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 along with the effect of the step on the memory (and other non stateless devices, if those exist). Note that the entire execution can be defined by the first configuration (for achieving self-stabilization usually assumed arbitrary) and the external inputs at the clock ticks.

Additional necessary and sufficient hardware support: 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, where the state of the processor, when the first fetch starts is arbitrary. (Means for achieving such a behavior are presented in [7]).

We assume there is a *watchdog* device connected to the NMI connector which is guaranteed to periodically generate a signal every predefined time. Watchdog devices are standard devices used in fault-tolerant systems e.g., [5, 11]. We have to design the watchdog to be self-stabilizing as well. The watchdog state is, in fact, a countdown register with a maximal value equal to the desired interval time. Starting from any state of the watchdog, a signal will be triggered within the desired interval time and no premature signal will be triggered thereafter. The watchdog guarantees execution of a critical operating system code such as code refresh and consistency checks [11], as well as memory management operations addressed in this work.

In order to guarantee that the processor will react to an NMI *trigger*, we suggest the addition of an internal countdown register or NMI *counter* as part of the processor architecture. This NMI counter will be decremented in every clock tick until it reaches zero. Whenever an NMI handler is executed (the processor can detect this according to a predefined program counter value), the NMI counter is raised to its maximal value (chosen to be a fixed value greater than the expected execution length of the NMI handler). The processor does not react to NMIs when the NMI counter does not contain zero. In addition, the `iret` operation assigns zero to the NMI counter. Thus, we guarantee that NMIs will eventually be handled from any processor state. In addition, while one NMI is handled, all other NMIs will be masked. Eventually, however this masking will be discarded in order to allow the next NMI. We say that a processor is in an NMI *state* whenever the NMI connector is set and the NMI counter contains 0, which means that the next operation to be executed is the first operation of the NMI handler procedure¹.

A read only memory should be used for storing fixed values. Specifically, the ROM will contain at least the interrupt table entry for the NMI and the NMI handler routine. This is needed in order to guarantee the execution of the NMI interrupt handler which, in turn, will regain consistency.

Figure 2(a) illustrates the legal execution of the system. The system is composed of various processes all of which execute in their turn. Additionally, there is a scheduler which is part of the NMI handler. The scheduler

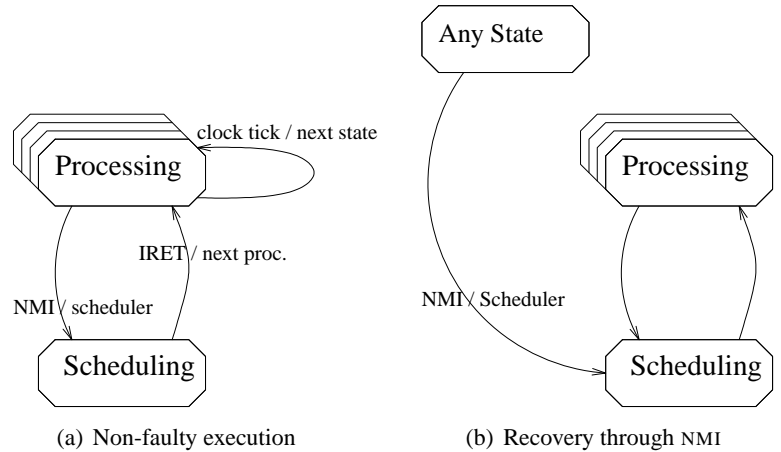


Figure 2: System Transitions

¹Note that the Pentium design has a similar mechanism that ensures that no NMI is executed immediately after an `sti` instruction.

repeatedly establishes its own consistency and also carries the process switch operation. It then validates the next process' state and sets the program counter so that the next chosen process will be executed. Due to a fault, the system may reach any possible state, as seen in Figure 2(b). However, due to the NMI trigger design, eventually the scheduler code will be called and will establish the required behavior.

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

The memory manager requirements include both the **consistency** and the **stabilization preservation** requirements:

Consistency: as the system executes, the memory manager keeps the memory hierarchy consistent (analogously to the consistency requirement for non-stabilizing operating systems). Namely we have to show that the contents of say, the main memory and the disk are kept mutually consistent.

Stabilization preservation: the fact that process and scope switching occurs, and that the memory is actually shared with other processes, will not falsify the stabilization property of each of the processes in the system.

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

3 Total Swapping — One Process at a Time

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

The settings for this solution are: N code portions, one for each process in the system, reside in a persistent read only secondary storage. The soft state of each process is repeatedly saved on the disk. The operating system includes a self-stabilizing scheduler (discussed in [11]), which activates processes in a round robin fashion. Whenever a process is activated, the process has all the memory for its operation (except the portion used by the scheduler). The scheduler actions include saving the state of the interrupted process in the disk and loading the state of the new process whenever a process switch occurs.

The scheduler executes a process switch at fixed time intervals². The processor state (register values) is saved in the stack. Note that we ensure that for every processor state, stack operations will not prevent the execution of the NMI handler and that the scheduler code will be started. This is needed since, according to the processor architecture, part of the processor state is automatically saved to the stack (during a context switch). These automatic stack operations carry the risk of unplanned exceptions. Thus, we ensure that whatever the stack state is, the handling procedure can be eventually called.

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

²Note that a clock interrupt counter may form a self-stabilizing mechanism for triggering a process switch, and that the counter upper bound is achieved regardless of what the counter value is when counting begins.

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

```

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

```

Figure 3: Total Swapping Algorithm

Correctness proof:

We use similar settings and proof style as in [11], so the reader may choose to read [11].

Lemma 3.1 *In every infinite system execution E , the program counter register contains the address of the swapping procedure's first instruction infinitely often. Additionally, the code is executed completely and every process is executed infinitely often.*

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

Lemma 3.2 *In every infinite execution E , memory consistency requirement holds.*

Proof: In every context switch, the whole data portion of a process is saved to the disk and is reloaded when needed again. The addresses used for these transfers are calculated every time and are based solely on the process number. The state of the scheduler is actually only the process pointer value which is incremented in every execution and is validated to be in the range 1 to N (the process table entries are, in effect, part of each process' state). Thus, even a corrupted value of a process pointer that may cause loading or saving the wrong data for a process, will not falsify the execution of the scheduler. Consequently, next time the effected process is loaded, the process will have the correct code (thereafter, a self-stabilizing process will converge). Based on Lemma 3.1, the above is true in every infinite execution. ■

Lemma 3.3 *In every infinite execution E , stabilization preservation eventually holds.*

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

Corollary 3.4 *The total swapping memory manager is self-stabilizing.*

Details of the implementation for this solution appear in Appendix A.

4 Fixed Partition — Multiple Residing Processes

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

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

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

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

The algorithm `Find-Frame` searches the frame table for a free frame. In case all frames are used, a particular frame is chosen for replacement. First the frame currently pointed to by this process' entry is validated to be in

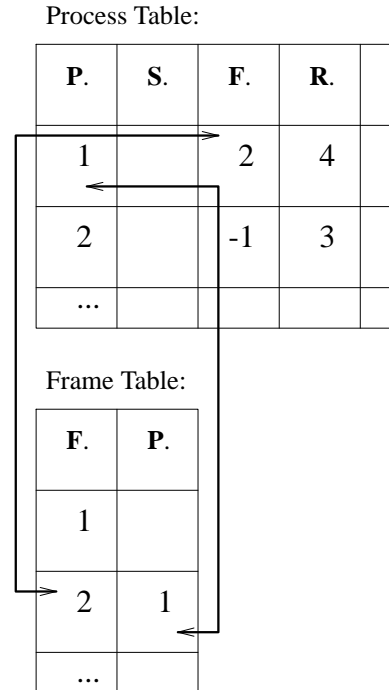


Figure 4: Fixed Partition Consistency Check

```

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

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

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

```

Figure 5: Fixed Partition Algorithm

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

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

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

The correctness of the algorithm is based on the ongoing consistency checks of FT and PT . Figure 4 demonstrates the consistency check made when assigning a frame to a process. Frame 1 is assigned to p_2 . Thus 1 is entered in the 2nd entry of FT . Additionally, the frame field in the entry of p_2 in PT (column marked with F) is marked with the new frame number. The arrow lines demonstrate the exclusive ownership of the selected frame for every scheduled process. Additionally, the refresh field (column marked with R) shows the refresh counter which ensures the periodical refreshing of the code for the processes. (The S column represents the processor state for each process).

Correctness proof:

Lemma 4.1 *In every infinite system execution E , the program counter register contains the address of the memory*

management procedure's first instruction infinitely often. Additionally, the code is executed from the first instruction (line 1 of Figure 10) to the last instruction (line 42) infinitely often.

Proof: The arguments are essentially the same as in Lemma 3.1. The procedure `Find-Frame` (of Figure 5) for finding a new frame for the running process is the only one containing a loop. This loop is bounded to run no more than M times, which happens when the whole frame table is scanned for a `NIL` value. If the code is executed without validation of the limit parameter (line 1 of `Find-Frame`) and without the advancing of nf , the new frame pointer, which limits the loop execution (line 2), the NMI mechanism will enforce an execution from the first line. This in turn, will be followed by the examination of the loop conditions and will ensure correct execution. ■

Lemma 4.2 *In every infinite execution E , the memory consistency requirement holds.*

Proof: Based on Lemma 4.1 every process will be executed infinitely often and the memory management algorithm will be executed prior to the execution of the process. The memory manager will cause the correct code for such a process to be loaded infinitely often from the stable storage since the refresh counter is decremented infinitely often. Therefore, the refresh counter will reach the value zero, which will cause code reloading. The correctness of the reloaded code is based on the direct mapping between the process pointer i and a disk location (as in Lemma 3.2). Secondly, the target address in the main memory (frame) is validated each time to be exclusively owned by the running process. Suppose that due to a fault, two processes p_i and p_j are marked in the process table PT as residing in the same frame number f in the main memory. Whenever the scheduler activates the first process, say p_i , it first validates (in line 1 of Figure 5) that the f entry in the frame table FT is i . If not, a new frame will be selected for p_i and both PT and FT will be updated accordingly (line 4-5 of `Swap-Process`). Even if the new selected frame is still f , the frame table entry for f will now contain i . Thus, when p_j will be scheduled, the memory manager will detect that a new frame should be chosen for p_j . ■

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

Lemma 4.3 *Stabilization preservation eventually holds.*

Proof: Each process eventually resides in the correct frame by Lemma 4.2. In addition, the applications must refer to main memory addresses in the frame size only (the implementation relies on the segmentation mechanism of the processor). Also, the code is fixed and does not change the content of the segment registers (note that such a restriction can be imposed by a compiler). Additionally, the correctness of the segment register assignment is repeatedly checked by the memory manager. Thus, the processes are effectively separated one from another, and the execution of one process can not alter the state of another process. ■

Corollary 4.4 *The fixed partition memory manager is self-stabilizing.*

Details of the implementation for this solution appear in Appendix B.

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

5 Dynamic Allocation

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

A memory manager process is responsible for allocating and for memory garbage collection. The dynamic memory manager uses bookkeeping for managing the dynamic memory allocations. Allocations are tracked using a table that holds the number of the owner process and the remaining lease period for each allocation unit. The dynamic memory manager repeatedly checks for memory portions allocated to a process for which the lease expired, and returns every such memory portion to the available memory pool for reallocation. The lease policy leaves the responsibility for refreshing the leases to the programmer of the processes, and at the same time allows simple and stabilizing dynamic memory management. We can argue that starting in an arbitrary configuration, where the dynamic memory is allocated randomly to processes, eventually no memory will be allocated to a process which did not request memory (recently). Moreover, assuming no malicious process behavior in every infinite execution, repeated allocation requests will be infinitely often respected. Up until this solution, programs were totally ignorant of operating system services. Here the operating system exposes an application programming interface for memory requests. Thus, programs should now also deal with temporary rejections of requests while the operating system makes sure that eventually all legal requests will be respected. The algorithms described below address the issue of dynamic allocations. Other needed mechanisms, like the automatic refreshing of code, are taken from the previous solutions.

Figure 6 presents the algorithms which implement the interface which programs can call in order to use dynamic memory. The MM-Alloc procedure is used for requesting memory allocation. With MM-ExtendLease a lease extension is possible. The applications are restricted to using the allocated memory through a special segment selector register and the procedure MM-NextSegment is the only way of accessing the different segments allocated to an application. At last, applications can release their allocations with MM-Free. The operating system contains a specialized process called `_MM-Validator`³ that validates the system's state concerning dynamic allocation. The algorithm is presented in Figure 7. Additionally, we use several service procedures which are presented in Figure 8.

Next, we describe how the algorithms work and consequently argue concerning their correctness. The MM-Alloc algorithm inputs are the number of allocation units (segments) required by the process and the expiration period needed. The expiration is the number of activations of the process for which the allocation will be valid. This number is bounded (at least) by the parameter length. After this period, the validator will reclaim those segments and mark them as free. In line 1 of the algorithm, the *dynamic selector* (which in the implementation is realized in a specific processor segment register) is checked for holding an empty address. If this is not the case, the meaning is that this process is already using dynamic memory and that the request is rejected in line 2 (for simplicity reasons we allow only one allocation at a time). In line 3 we check whether there are sufficient allocation units for this request through a global variable that holds this count. We assign the requested quantity to the requesting process with the `_MM-Assign` procedure which simply goes over all the segments in the segment table ST and marks the needed quantity as occupied. This procedure also updates the free segment variable (line 5), and sets the dynamic selector value with the address of one of the allocated segments (line 9). In case insufficient

³The leading underscore marks a procedure internally called by the operating system

```

MM-ALLOC(quantity, expiration)
1  if seg(PT[currentProcess])  $\neq$  NIL
2  then return
3  if quantity  $\leq$  freeSegments
4  then _MM-ASSIGN(currentProcess, quantity, expiration)
5  _MM-ENQUEUE(currentProcess, quantity, expiration)

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

MM-NEXTSEGMENT()
1  currentSegment  $\leftarrow$  seg(PT[currentProcess])
2  if currentSegment  $\neq$  NIL
3  then for each s in  $\{(currentSegment + 1) \text{ modulo } NUM\_SEG..$ 
4   $(currentSegment - 1) \text{ modulo } NUM\_SEG\}$ 
5  do if owner(ST[s]) = currentProcess
6  then seg(PT[currentProcess])  $\leftarrow$  s
7  break

MM-FREE()
1  currentSegment  $\leftarrow$  seg(PT[currentProcess])
2  MM-NEXTSEGMENT(currentProcess)
3  if currentSegment  $\neq$  NIL
4  then if currentSegment = seg(PT[currentProcess])
5  then seg(PT[currentProcess])  $\leftarrow$  NIL
6  if owner(ST[currentSegment]) = currentProcess
7  then owner(ST[currentSegment])  $\leftarrow$  NIL
8  freeSegments  $\leftarrow$  freeSegments + 1

```

Figure 6: Dynamic Allocation Services

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

The *MM-ExtendLease* procedure carries out its task by validating that the requested segment is owned by the requesting process and enlarges the lease counter value. Again, this operation is allowed assuming there is no a malicious behavior of processes. A different approach can enable the extension just in cases when the queue is empty, thus preventing a repeated extension of a lease by a particular process. As mentioned before, a process can access the allocated segments through a selector which it cannot change. In order to move between allocated segments, the process calls *MM-NextSegment* which looks in the segment table for all other segments and if another segment is also occupied by the calling process, its number is returned by the selector (line 6). The *MM-Free* procedure carries out its task by first updating the selector with another segment address (lines 1-2). It then checks if this selector is the only one owned by this process, which means that the selector should be cleared too (line 5). In lines 6-8, the released segment is checked for being owned by the process and is consequently marked as free. The global counter of free segments is updated respectively.

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

```

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

```

Figure 7: Dynamic Allocation Validation

the lease for each of them. In case a lease reaches zero, the segment is marked as free (line 9). Otherwise, we mark the process as using dynamic memory (line 10). Lines 13-15 reset the dynamic selector (saved in the process table PT) for processes that do not currently use dynamic memory. Then, we check the queue top and in case the first waiting process can be satisfied with the current free segments, it is deleted from the queue (line 18) and assigned with the free segments (line 19). The $_MM-Dequeue$ procedure merely moves all the entries in the array having the queue progress one cell towards the queue top. It also marks the last entry as free.

Correctness proof:

Lemma 5.1 *In every infinite system execution E , the program counter register contains the address of the validator procedure's first instruction, infinitely often. Additionally, the validator code is executed entirely.*

Proof: Based on Lemma 3.1 the scheduler schedules every process of the system infinitely often. and specifically the validator process. Note that the scheduler also checks that the program counter value is within the fixed limits of the program size. The code of the validator process does not contain branches out of the the procedure limits. All the contained loops are **for**-loops with advancing index which is checked each time for being inside fixed limits. Since this index advances each time, every loop is guaranteed to reach the end criteria and have the program counter advance towards the procedure end. ■

Lemma 5.2 *In every infinite system execution E , eventually no memory will be allocated to a process which did not (recently) requested it.*

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

Lemma 5.3 *In every infinite system execution E , repeated allocation requests will be respected infinitely often.*

```

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

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

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

```

Figure 8: Dynamic Allocation Service Procedures

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

Lemma 5.4 *In every infinite execution E , the memory consistency requirement holds.*

Proof: The segment table *ST* records the owning process for each segment. Even if the table is transiently corrupted, based on Lemma 5.2, eventually only requesting processes will be marked as using a segment. Thus, no other process is using this memory. The access to each segment is only via a segment register. The value of which we assume cannot be changed by the process. This mechanism enforces the usage of this segment only by the marked process, achieving memory consistency. ■

Lemma 5.5 *In every infinite execution E , stabilization preservation eventually holds.*

Proof: In this case as well there is a full separation of memory accesses of processes. The static areas are separated by means of the segment selectors from the previous solution while the dynamic areas are separated based on Lemma 5.4. Thus, every stabilizing process will stabilize in spite of the actual sharing of memory with other processes. ■

Corollary 5.6 *The dynamic allocation memory manager is self-stabilizing.*

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

Details of the implementation for this solution appear in Appendix C.

6 Concluding Remarks

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

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

We have run the presented system using the BOCHS [2] simulator. During some of the executions we completely changed the contents of the RAM and observed that stabilization was achieved. Namely, the processor eventually continues to execute the correct code of the operating system.

We are convinced that self-stabilizing operating systems will be part of every critical computing system in the near future. Prototype implementations can be found in [23].

References

- [1] O. Brukman, S. Dolev, H. Kolodner. "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software", *Proceedings of IEEE International Conference on Software-Science Technology & Engineering*, (Sw-STE03), Israel, 2003.
- [2] Bochs IA-32 Emulator Project.
<http://bochs.sourceforge.net/>
- [3] L. A. Belady, R. P. Parmelee, C. A. Scalzi. "The IBM History of Memory Management Technology", *IBM Journal of Research and Development* 25(5), pp. 491-504, 1981.
- [4] M. Baker, M. Sullivan. "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment", *Proceedings of the Summer 1992 USENIX Conference*, Texas, June 1992
- [5] 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, CA, October 2000.
- [6] R. C. Daley, J. B. Dennis. "Virtual memory, processes, and sharing in Multics", *Proceedings of the first ACM symposium on Operating System Principles*, p.12.1-12.8, January 1967, Gatlinburg, TN.
- [7] S. Dolev, Y. Haviv, "Self-Stabilizing Soft Error Resilient Microprocessor" *17th International Conference on Architecture of Computing Systems*, LNCS:2981, (ARCS04), 2004. Also to appear in *IEEE Transaction on computers*.
- [8] E. W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM*, Vol. 17, No. 11, pp. 643-644, 1974.
- [9] S. Dolev. *Self-Stabilization*, The MIT Press, Cambridge, 2000.

- [10] B. Demsky and M. Rinard. “Automatic detection and repair of errors in data structures”, *Technical Report MIT-LCS-TR-875*, MIT, 2002.
- [11] S. Dolev, R. Yagel. “Toward Self-Stabilizing Operating Systems”, *Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS04,DEXA)*, pp. 684-688, Zaragoza, Spain, August 2004.
- [12] T. Herman, T. Masuzawa. “Available stabilizing heaps”, *Inf. Process. Lett.* 77(2-4), 2001.
- [13] IBM. Autonomic computing initiative,
<http://www.research.ibm.com/autonomic>, 2001.
- [14] Intel Corporation. “The IA-32 Intel Architecture Software Developer’s Manual”,
<http://developer.intel.com/design/pentium4/documentation.htm>, 2005.
- [15] M. Kistler, P. Shivakumar, L. Alvisi, D. Burger, and S. Keckler. “Modeling the effect of technology trends on the soft error rate of combinational logic”. In *ICDSN*, volume 72 of *LNCS*, pages 216–226, 2002.
- [16] B. W. Lampson. “How to build a highly available system using consensus”, *Distributed Algorithms*, *LNCS 1151*, 1996.
- [17] 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.
- [18] G. Muhl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, A. Ulbrich. ”Self-stabilizing publish/subscribe systems: Algorithms and evaluation”. *Proceedings of the 11th European Conference on Parallel Processing (Euro-Par 2005)*, (*LNCS 3648*), 2005.
- [19] The Netwide Assembler.
<http://nasm.sourceforge.net>.
- [20] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. “Recovery Oriented Computing(ROC): Motivation, definition, techniques and case studies”, UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002.
- [21] Jerome H. Saltzer. “Protection and the control of information sharing in multics”, *Communications of the ACM*, v.17 n.7, p.388-402, July 1974.
- [22] M. M. Swift, B. N. Bershad, H. M. Levy. “Improving the reliability of commodity operating systems”, *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP’03*, Bolton Landing, NY, October 2003.
- [23] <http://www.cs.bgu.ac.il/~yagel/sos>
- [24] <http://www.selfstabilization.org>
- [25] Sun Microsystems, Inc., “Predictive Self-Healing in the Solaris™10 Operating System”, White paper
http://www.sun.com/software/whitepapers/solaris10/self_healing.pdf, September 2004.

A Total Swapping Implementation

The implementation for the total swap solution appears in Figure 9. The code resides in ROM and is executed following an NMI trigger. The code for saving and loading the processor state and for incrementing the process pointer is an extension of the one presented in [11]. Therefore equivalent parts are omitted here.

The code contains no loops, and all address calculations are based solely on the value of i which points to the relevant process. Note that all values are also recalculated for every execution. Namely, every variable's first appearance following line 1 is associated with a command that loads a value. The only exception is the variable i , which is verified to be in the range 1-N in lines 1-2. The rest of the code contains three stages for calculating the required parameters for the disk routines. Lines 3-10 save the process state to disk. Immediately after these lines the process number i is incremented and following that, lines 11-18 load the code for the next process from the read-only stable storage, while in lines 19-26 the state of the process is loaded from disk. Afterwards, the processor state for the process is also loaded and the process is finally activated. In more details, lines 3-4 load the register ex with a fixed main memory address, pointing to where the running process is residing. Line 5 loads register bx with another fixed address which is the offset to the location of the process state.

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

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

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

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

Figure 9: Total Swapping Implementation

separated by this full swapping algorithm, the whole system will eventually stabilize too.

B Fixed Partition Implementation

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

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

Figure 10: Fixed Partition Implementation

Finding a new frame for the new process involves validating the current frame value of p_i (line 16) and then increasing this value modulo M (lines 17-18) in order to start the search. Yet, these operations are only based on the correct assignment to register bx made by the scheduler. Lines 19-20 check if the search ended by moving across the whole of the frame table, while lines 21-25 check in FT whether an empty frame is reached. Note that the check concerning the occupation of a frame by a non-active frame (line 4 of the pseudo-code in Figure 5) is omitted here. This does not violate the correctness of the algorithm, but might cause unnecessary swapping of frames. Lines 26-28 increment the new frame counter modulo M and jump back for checking the loop end conditions, as mentioned above. Upon exiting (line 29), the register al contains the value of a new frame for the scheduled process.

In order to actually swap out a residing process p_j , we first check that the relevant new frame nf is not marked empty. This is done in lines 30-34 in the same way as before by accessing FT . If nf is not empty, then p_j 's state is saved to disk in line 35. The procedure that writes to disk uses the values of nf and j . In case the latter value is corrupted in FT (j is taken from FT), the state of p_j will be corrupted, since the state of the process residing in frame nf will be saved instead of the state of p_j . The process p_j will be scheduled again later, will be loaded with its correct code and will stabilize. For better bookkeeping, the frame field for p_j in PT should be marked NIL (line 3 of pseudo code in Figure 5). This is not necessary for correctness, since when rescheduling p_j a contradiction between PT and FT will be found and resolved. Thus, it was omitted from the current implementation. Lines 36-37 update the nf entry of FT with the new residing process number i (addresses based on the values of i which are maintained by the scheduler and register si , calculated in line 30). Line 38 updates the i th entry of PT with the correct frame nf and consequently the state of p_i is finally loaded from disk (line 39). The parameters for this procedure are only i (maintained by the scheduler) and nf , both of which were calculated before. Line 40, implements the decrement of the refresh code countdown, and is also based only on the bx value. Line 41 returns from the swap algorithm.

C Dynamic Allocation Implementation

Figure 11 presents the implementation of the garbage collection algorithm of Figure 7. Lines 1-3 enforce correct values of segment registers (ds , es) for correct transfer of data. Lines 4-8 carry the task of zeroing the array in memory which marks the processes that are actively using dynamic memory. This is done by loading the memory address in register di , preparing a zero in register ax for copying (actually only the lower half is used), and loading the size of the array into register cx . Line 7 assures the correct increment of the destination address pointer, by setting the direction flag of the processor. Thus, the whole state is validated towards the actual copy performed in line 8. The `rep` instruction causes the processor's microcode to perform a loop which decrements register cx towards zero. In line 9 we assign the variable counter of free segments with zero, It will later be incremented for each segment which will be found free. The final value will be used for deciding whether to assign the free memory to the next waiting process.

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

In case the lease is still valid, lines 18-21 mark the process as using dynamic memory. This is done by taking the base address of this array and adding the value stored in register `al` which was assigned in line 12 to the holding process number. In case the lease expired, the segment is marked free by assigning a special NIL value at the owner field (line 22). Then, in lines 23-25 we check for such a NIL value (whether achieved by the lease reduction or by a process explicitly releasing a segment), and updating the free segment counter accordingly. Lines 26-27 increment the pointer to the segment table and the loop counter, and as long as it is not zero, jump back to line 12.

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

The procedure ends with an attempt to allocate the available dynamic memory to a waiting process. Line 38 points register `si` to the memory holding the queue. This area contains requests for memory allocation containing the requesting process id, the requested quantity and a lease period. In lines 39-41 we check whether the queue is empty and if it is, there is no more work to be done. Lines 42-45 check whether the top request size is larger than the available segments. If that is the case there will be no further action (until additional space is freed).

Note that line 43 validates that the request size is not larger than the whole dynamic memory size.

```

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

```

Figure 11: Dynamic Memory Validator Implementation

Otherwise, an error in that value might prevent all future allocations (this argument does not hold for the lease parameter which is designed to take care of any possible value. This value is bounded only by the size of the containing variable). Finally in lines 46-48 we are in a state in which a new allocation can be made. The lease expiration value is prepared in register `ch` and we call the two utility procedures which pop the request from the queue and assign the necessary free memory.

Figure 12 presents the implementation for the `MM-Alloc` procedure, called by the processes in order to obtain dynamic memory, and also the internal `MM-Assign` procedure. (The other routines e.g., `MM-Free`, that are mentioned in the pseudo code, are simpler to implement and are mainly used for performance optimizations, therefore were omitted from the current implementation version). Figure 13 completes the implementation with the queue handling procedures. The code corresponds directly to the pseudo code algorithms presented in Section 5. Lines 1-3 of `MM-Alloc` insure that the requesting process does not hold any dynamic memory already any more. This is ensured by checking the `fs` register which only points to a dynamic segment address for granted processes. Lines 4-6 prepare the required parameters for calling the `MM-Assign` procedure (line 9) and check if substantial memory is available. Otherwise, the enqueue procedure is called for in line 7. The assignment procedure sets up a loop (lines 11-13) for traversing the memory segment list. Then, each list entry is checked for emptiness (lines 14-19). In case an empty entry is found, it is marked as owned by the requesting process and the lease is recorded too (lines 20-21). Line 22 decrements the global counter for free segments. Then, in lines 23-25 a check is carried out in order to see whether enough allocations had already been made. Otherwise the loop continues through lines 26-27. Lines 28-30 update the segment register pointer with the new allocation. Since this procedure can also be called independently by the validator process (lines 31-36), this update is carried to the process' state in the process table too.

The `MM-Enqueue` operation of Figure 13 works as follows. In lines 1-10 the queue is searched for the requesting process. If found, the operation stops in line 7. In lines 11-23 an empty slot is searched for placing the request. The `MM-Dequeue` operation is carried out by traversing the queue (lines 25-37) and by advancing each slot by one location. In order to mark it as free, line 38 places a `NIL` value at the end of the queue.

```

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

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

```

Figure 12: Dynamic Memory Implementation

```

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

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

```

Figure 13: Dynamic Memory Helper Procedures