M-M/S-CD Memory Management for Second Generation Microkernels

Ya. I. Klimiankou

The paper introduces a new memory management approach for second generation microkernels named M-M/S-CD that was designed in the spirit of minimality principle. It was developed from scratch based on analytical model of computer memory system. M-M/S-CD memory management pushes out all memory management activities and policies into the user mode applications, where kernel only enforces two conditions are met that assure that memory system will remain in closed state.

Keywords: second generation microkernel, memory management

Introduction

Microkernel operating system design currently is still the most promising. Invented by the Mach project [1], microkernel design became very popular in scientific society due to the number of significant advantages which it offers. Mach opened a series of first-generation microkernels built in accordance with principle claiming that kernel must be the only software module executing in privileged mode. Second generation of microkernels was invented by Jochen Liedtke L4 project, which added the minimality principle to the microkernel design. Minimality principle states : "A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality" [2]. According to this principle, kernel must be kept as small and simple as possible.

Memory management (MM) is one of the major and unavoidable functions of the operating system (regardless of its design). MM is a process of planning, organizing and controlling of computer memory use. In modern computer systems, MM is performed in terms of a physical address space (PAS) and a virtual address space (VAS), and can be divided into two disciplines: physical memory management and virtual memory management. Memory management can't be completely moved from kernel to user space because memory is one of the most important system resources and it requires access to the privileged CPU-provided facilities.

A number of second generation based microkernel operating systems has been proposed, each of which addresses memory management in different ways. There are two commonly adopted approaches: recursive address space construction [3, 4, 5] and capability-based memory management [6]. We believe that both are still overcomplicated in regard to the minimality principle. The purpose of this paper is to present another one approach to the memory management in context of second-generation microkernels.

Computer system memory can be represented by two entities: physical space (PS) and virtual space(VS), where the first one is a set of all physical pages in the system and the second one is a set of all virtual pages in the system. These two sets can be further subdivided into four sets: physical pages mapped to the external resources (PMP) and not mapped (PHP), virtual pages mapped to the external resources (VMP) and not mapped (VHP). According to this, memory management can be considered as an establishing of the mapping between physical space and virtual space:

$$\begin{gathered} PS \mapsto VS \\ (PMP,\ PHP) \mapsto (VMP,\ VHP), \end{gathered} \tag{1}$$

where two conditions must be always met:

$$\forall P_i \in PMP,\ \exists(P_i \mapsto P_j),\ P_j \in VMP \tag{2}$$

$$\forall P_i \in PHP,\ \nexists(P_i \mapsto P_j),\ P_j \in VMP, \tag{3}$$

where P is a memory page (either virtual or physical, depending to which address space it belongs).

The first condition (2) is referred as memory leakage prevention. In context of high level languages memory leak refers to keeping a memory block in use after finishing its actual use. It can lead to exhaustion of available system memory or available address space. In contrast to high level languages, memory leak in the context of operating system memory management refers to exclusion of a memory block from the pool of memory in use. Under the normal conditions it is expected that all available memory of the system must be used to produce value and lost of last reference to the memory block prevents its future use for value production. Due to this operating system must enforce memory management system to keep at least one reference (mapping to virtual memory page) for each physical memory page linked to external resources.

The second condition (3) is referred as access violation prevention. As it was mentioned above both PAS and VAS can include pages that are not linked to external resources. But processor behaves differently on attempts to access memory in page that isn't linked to any external resources in address spaces of both types. Access to such page in PAS layout is almost completely invisible both for application programs and for operating system. Data write to physical page not linked to external resources will be ignored and data read from this page will return to the reader an undefined value captured from the data bus. However, in both cases, neither error signal will be generated by processor logic nor any other handling of invalid access will be made. In contrast, CPU can detect access to the virtual pages not mapped to physical pages and trigger exception in reply. This exception usually invokes operating system to correctly handle invalid memory access. As a result VAS can be used to protect software from access to the PAS pages not linked to external resources. Access violation prevention assumes enforcing of the rule according to which no one page of the PAS not linked to external resources must ever be mapped to any VAS page. Enforcing two conditions described above is enough to turn the computer memory system in the closed consistent system state. Due to it enforcing these two conditions of memory management is the only activity that must be done on the operating system kernel side. The rest of memory management activities and policies can be safely implemented on the side of memory managers implemented in user mode, which are free to manipulate mappings between PAS and VAS pages until the conditions described above are hold in the true state.

M-M/S-CD Memory Management

There are two main requirements applied to the memory management system described in the analytical model above:

- memory leakage prevention;
- access violation prevention.

Access violation prevention can be satisfied by providing guarantees from the kernel design, according to which the initial PAS layout provided by third party provider will be kept in the constant state. It means that no pages will be added or removed to\from the PAS layout during the system lifetime. It is assumed, that third party provider (for example system bootstrap module or node bootstrap module) must deliver to the kernel a set of initial VASes consisting exclusively from mappings to PAS pages associated with backend resources like RAM, ROM, memory-mapped IO device registers. There must be no mapping to the PAS page belonging to the PAS layout hole.

In contrast to the classical memory leakage problem that is usually considered in the context of high-level programming languages, the memory leakage in the context of kernel memory management means the loss of the last mappings to the particular PAS page, and thus loss of the ability to incorporate that PAS page in any VASes for future actual use by applications. Satisfaction of memory leakage prevention requires explicit enforcing of memory management restrictions on the kernel side, because associations between PAS pages and VAS pages can be established and terminated dynamically, which in turn is true because VASes can be created and destroyed by kernel in run time and VASes pages associated with VAS layout holes can be converted into the VAS pages associated with actual PAS page memory and vice versa. Due to this kernel must enforce the policy according to which it will be impossible to lose the last mapping to the PAS page.

Both requirements described above can be satisfied by applying the memory management model that is referred as M-M/S-CD. This model suggest dividing of all VAS pages present in system into two classes: master pages and slave pages. Each PAS page associated with backend resource is linked with exactly one master VAS page, and thus that master page represents this PAS page in the system. M-M/S-CD memory management suggests that the master page can not be created or removed directly by kernel. The set of master pages is delivered to kernel by third party component, such as boot loader, as part of kernel initialization process and stays constant during the whole system lifetime. The only action that can be applied to the master page is moving between different page slots in the same VAS or between different VASes present in a system. In contrast, slave pages have dynamic nature and can be created from the master page and destroyed multiple times over the system lifetime. There can be multiple slave pages spawned by the same master page at the same moment of time. As a result, according to the M-M/S-CD memory model computer memory system is considered to be a closed system in relation to master pages, which in their turn represent PAS pages associated with memory (see Figure 1).

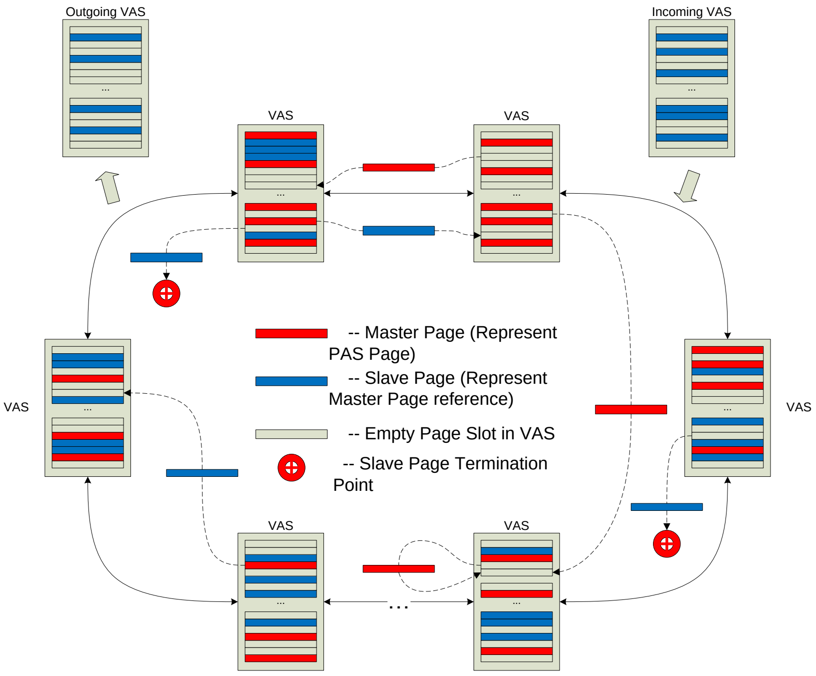
Outgoing VAS
Incoming VAS
VAS
VAS
VAS
VAS
VAS
VAS
-- Master Page (Represent PAS Page)
-- Slave Page (Represent Master Page reference)
-- Empty Page Slot in VAS
-- Slave Page Termination Point

Figure 1. M-M/S-CD Closed Memory System

The described M-M/S-CD model of a closed system of system memory implies only three conceptual operations on memory pages:

- Moving of master page (*svas*, *spfn*, *tvas*, *tpfn*) - atomic operation that removes master page from the *spfn* page slot from the *svas* virtual address space and inserts it in the empty *tpfn* page slot in the *tvas* virtual address space.
- Creation of slave page (*svas*, *spfn*, *tvas*, *tpfn*) - makes a copy of master page from the *spfn* page slot from the *svas* virtual address space and inserts that copy converted to a slave page in the empty *tpfn* page slot in the *tvas* virtual

address space.

- Destruction of slave page (*svas*, *spfn*) - removes the slave page from the *spfn* page slot from the *svas* virtual address space.

To distinguish different page types in our prototype implementation on Intel x86 platform, two of the three available for use bits of the page descriptor were used. One of these bits is used to mark master pages and the second one is used for page slot lock, that is required at least for master page moving operation. Thus page slot has three bit flags: locked/unlocked, master/slave, present/absent.

Current Status and Conclusions

We developed the prototype of M-M/S-CD memory management facilities and incorporated them in our experimental OS. But we still have not any application level memory management service to check our solution in more real life environment.

Microbenchmarks showed that the overheads introduced by the basic memory management operations are 673, 765 and 864 cycles appropriately for the create, destroy and move operations. The empty system call introduces 373 cycles of the overhead. As a result, the overhead introduced by the M-M/S-CD exclusively is 300, 392 and 491 cycles for the create, destroy and move operations respectively.

To check the M-M/S-CD memory management approach and measure its overhead we plan to develop three user-mode memory managers (MM): paged MM, unpaged MM and file-mapping MM.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," in *Proceedings of the "Technical Conference – USENIX"*, pp. 93–112, 1986.

[2] J. Liedtke, "On μ-kernel construction," in *Proceedings of the "15th ACM symposium on Operating Systems Principles"*, pp. 237–250, ACM, 1995.

[3] A. Au and G. Heiser, *L4 User Manual. Version 1.14.* School of Computer Science and Engineering, University of NSW, 1999.

[4] K. Elphinstone, G. Heiser, and J. Liedtke, *L4 Reference Manual MIPS R4x00 Version 1.11.* School of Computer Science and Engineering, University of NSW, 1999.

[5] I. Kuz, *L4 User Manual. NICTA L4-embedded API. Version 1.11.* School of Computer Science and Engineering, University of NSW, 2005.

[6] S. Gerber, "Virtual memory in a multikernel," Master's thesis, Department of Computer Science, ETH Zurich, Switzerland, 2012.

Authors

Yauhen Ivanavich Klimiankou — the 2nd year postgraduate student, Faculty of Computer Systems and Networks, Belarussian State University of Informatics and Radioelectronics, Minsk, Belarus; E-mail: *Evgeny.Klimenkov@gmail.com*