Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp19-23)

Simulation Environment for Testing and Verifying the L4 Microkernel Mapping Database

M. K. REVUELTA, P. P. ESPADA, I. G. TEJEDOR, A. V. SÁNCHEZ
Departamento de Automática
Universidad de Alcalá
Escuela Politécnica, Campus Universitario, C.P. 28871 Alcalá de Henares
SPAIN

Abstract: - As the complexity of the operating systems increases, it has been proven necessary to develop new techniques to test and verify them. L4 is a small microkernel that serves as base for numerous research and development projects. One of its design paradigms is the externalization from the kernel of the user level memory allocation policies. This externalization forces the microkernel to maintain complex structures and use convoluted algorithms to process all the required information. Some of these structures and methods form what is called the mapping database subsystem. In this paper, we present a simulation environment that extracts the complete database subsystem from the rest of the kernel, and runs on it several automated tests and verification processes. With this simulation environment, several errors were discovered in early development stages of a new mapping database. A fast feedback loop of coding and testing soon led to a final version even more stable than the old, simpler, mapping database.

Key-Words: - Simulation, Validation, L4, Pistachio, Mapping Database, Test, Random

1 Introduction

Testing and verification of applications and systems has always been a challenge since the very first steps of the computer science development. Assuming that proving the complete absence of errors in operating systems and its applications is nearly impossible, several techniques have been used to minimize their errors. These techniques go from the simple running of different test programs to the formal verification using mathematical oriented languages [8].

In this paper, we present a new approach to the system testing and error detection problem. This solution has been actually used to test and verify the implementation of one of the key parts of the L4 Microkernel: the Mapping Database.

Basically, it consists in the extraction from the kernel of the mapping subsystem and the implementation of a parallel, easily-designed and well-proven system that simulates the behavior of the real one. This simulated system is implemented using some techniques that cannot be used in the real system, such as recursion, due to the limitation of the system resources. It is used to generate different scenarios and run several different operations in order to verify the correctness of the developing model. As we will see, the simulator itself is able to generate random tests, thus increasing the detection capabilities and reducing the mean error probability of the final system.

This testing methodology can be extrapolated to any other subsystem as long as it has enough independence from the rest of the operating system. It can also be used to develop and validate new computing algorithms and methods.

This paper covers the design of the mentioned simulation system developed for testing one of the key parts of the L4 Microkernel.

Section 2 briefly introduces the L4 microkernel and the Map/Grant/Unmap operations.

Section 3 introduces the mapping database, which supports the mentioned mapping operations. A quick glimpse to its complexity explains the motivation of the present work.

Section 4 presents the whole simulation system, with all its features and functionalities.

Section 5 exposes the results of the simulations.

Finally, section 6 summarizes the conclusions.

2 L4 Microkenel debriefing

L4 is a small microkernel originally designed and developed by Jochen Liedtke [4,5] at the GMD National Research Center. Nowadays, the project is maintained by the System Architecture Group of the University of Karlsruhe, the DiSy group from the University of New South Wales and the Technical University of Dresden. It follows a set of paradigms, which are a quick, non-complex, architecture efficient inter process communication (IPC) and

externalization of every service that does not require being executed in supervisor mode.

The kernel provides primary abstraction of threads and address spaces. The application interface is composed of a small set of system calls which offer basic services such as thread management, memory and communication primitives.

Most of the operating system services are externalized from the kernel, being granted by a set of user-level privileged threads called servers.

User level threads, including the servers, exchange information between them by using a complex set of communication mechanisms. During the process, the threads exchange the data through a set of virtual registers called message registers, whose implementation is architecture dependant.

L4 externalizes the user level memory allocation policies from the kernel. It defines an initial virtual address space called sigma0 (σ_0), which is the result of idempotentially mapping the complete physical memory. The rest of the defined virtual address spaces must receive mappings, directly or indirectly, from σ_0.

The kernel provides a set of three mapping primitives: Map, Grant and Unmap [6]. The first one is used to map a single block of virtual memory from one space to another. Grant is similar but with the difference that, in this case, the original memory mapping disappears. The last primitive can be used to cancel previously performed operations.

Figure 1 shows an example of the memory system. In this example, the physical memory remaining free after the booting process is mapped directly into σ_0 space and the rest of the virtual address spaces get mappings from it.

In order to preserve the coherency of the memory system, the kernel must maintain a set of structures with all the necessary information regarding the memory blocks mapped between the different virtual address spaces. This set of structures is called mapping database.

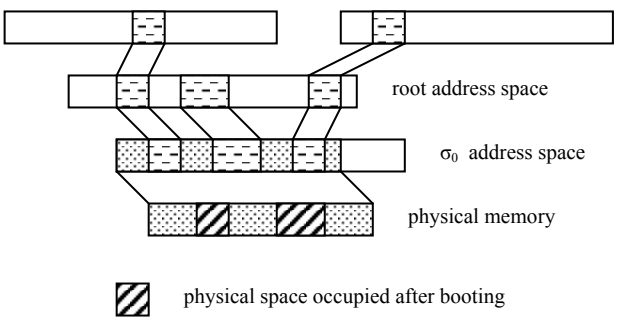

Figure 1. L4 virtual address spaces

The new mapping database, developed by Espen Skoglund, member of the L4Ka team, includes other cascade operations regarding access rights and accessed bits. All the references to the mapping database in this paper correspond to the new mapping database, available since February 2005 in the public CVS repository [2] of the L4Ka Team as part of L4Ka::Pistachio 0.4 [1].

3 The Mapping Database

The mapping database plays an important role in the L4 microkernel. It maintains a complex tree, storing all the currently valid mappings of the system. The tree structure is required to perform cascaded operations affecting all the mappings directly or indirectly derived from the starting node.

Every mapnode of the mapping tree represents a map operation from one space to another. The root of the tree represents the whole σ_0 space, which is an idempotent mapping of the physical address space.

Mappings can have different sizes. When a small mapping is established under a larger one, an intermediate data structure is used to keep track of the base address of the small mapping within the larger one. This intermediate data structure is itself a tree of arrays of rootnodes. Therefore, the mapping tree is not only a simple tree, but a tree of trees.

Figure 2 illustrates a mapping tree with mappings of different sizes. Mapnodes A and D represent mappings made directly from σ_0. Mapnode A is the same size as σ_0, but the rest correspond to smaller blocks of memory. The rootnodes arrays subtrees keep the smaller mapnodes organized depending on their offsets inside their parent mapnodes. Mapnode D is several levels smaller than σ_0, but it can reside

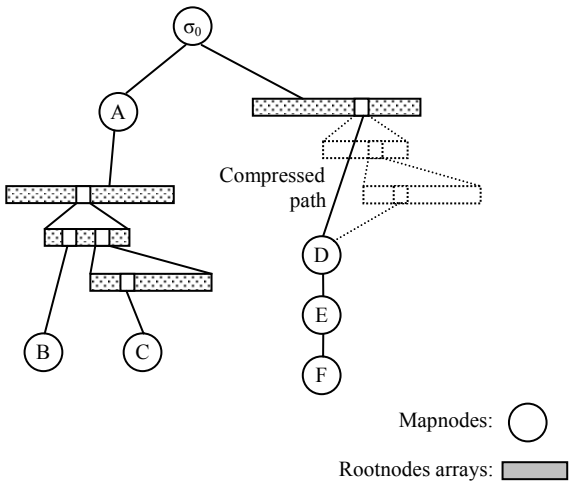

Figure 2. The mapping tree

under a larger rootnode entry saving intermediate rootnode arrays. This is called compressed path. Mapnodes E and F will be considered children of σ_0 or children of D, depending on an integer value called depth.

As any other part of L4, the Mapping Database is subject to dramatic restrictions regarding execution time, preemption and memory resources. Its heavily optimized implementation avoids ordinary recursion in order to keep the stack usage as low as possible. Only a limited amount of information requires being stacked, and it is stored in an array. The recursive travel through the tree is performed by loops instead of function calls. All operations can be quickly interrupted leaving the mapping database in a consistent state.

The implementation is additionally complicated as a result of the usage of other techniques like compressed paths in the trees of arrays of rootnodes.

4 Simulation process

The simulation process consists in extracting the mapping database and all its structures and functions from the microkernel, and encapsulating them in a complex simulated system.

Within this environment, the mapping database is tested in many different ways to detect possible conceptual malfunctions and/or implementation errors.

The C++ programming language contains a rich set of mechanisms that allow redefining data types with a very low impact on algorithmic parts of the code. It also includes the powerful substitution preprocessor directives inherited from C. These mechanisms are used in the mapping database to monitor its activity within the simulations. The mapping tree elements are wrapped by objects containing additional information and checkpoints are inserted in the most complex mapping operations. The validation is performed step by step. A similar approach has been previously suggested by [7].

Figure 3 shows the automated simulation process. The mapping database stands in the middle with a few modified lines of code that prevent infinite loops, which would stop the process. The surrounding elements will be explained along this section.

4.1 Dynamic memory subsystem

A dynamic memory subsystem provides the necessary memory for the different structures used in the mapping database. The subsystem assigns the blocks according to their size. This size must be also specified upon deallocation.

The simulated environment provides this kind of memory allocation system, but also traces the different assigned memory blocks, saving all the necessary information, such as the object type, its size and its address. With this tracing, the simulator verifies the assignments and deallocations of the blocks corresponding to the different structures.

After every step of the mapping process, the simulator checks that every used structure resides in an assigned memory block and that every assigned block is used to store a valid database object. It also checks that every object is attached to the mapping database tree.

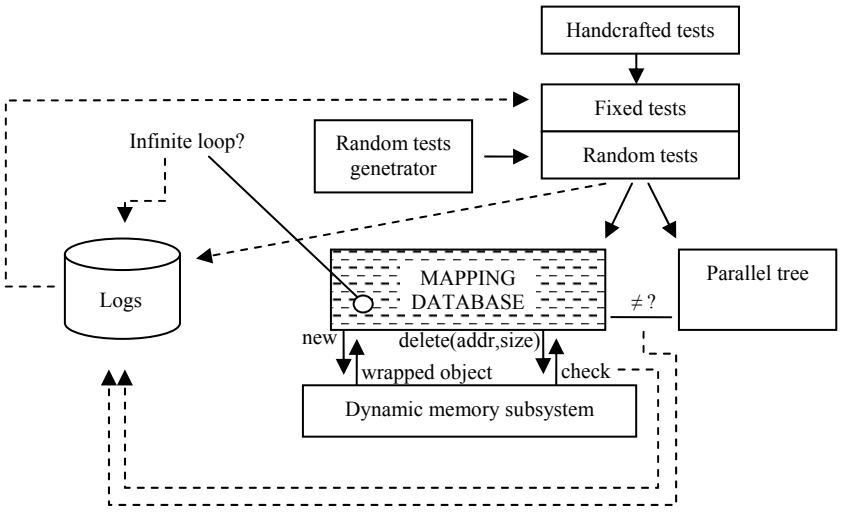

Figure 3. Simulation environment for the mapping database

4.2 Topology verification

Once an operation is performed in the database, the simulator travels through the whole mapping tree, verifying its topology. During this process, the simulator checks the correctness of the relationships between the different nodes of the database tree, validating their pointers.

4.3 Parallel tree

The simulation environment also implements a parallel tree that has the same features of the original one.

The main difference between the one provided by the simulator and the one from the mapping database is that the former uses, without any restrictions, all the normal resources provided by the operating system on which the simulator is run. It uses recursion and other memory-and-time-consuming mechanisms, not available in the heavily optimized version, implemented for the microkernel. Therefore, it is easier to ensure its correctness and it can be used as a reference model.

4.4 Run tests

The simulator runs two kinds of tests: predefined tests and random tests.

It is possible to define a series of predefined handcrafted tests that present different situations to challenge the mapping database model. These situations include mapping and granting operations of different memory blocks or changes in their respective access rights.

After the set of fixed tests, the simulator performs random actions on the database. In this case, the size of the mapped blocks and the access rights are randomly selected, allowing the detection of errors that would otherwise remain covered. The initial state for these tests is also built at random.

4.5 Log

The simulator stores precise information of the operations that are taken into a set of log files. The aim is to prevent loosing information due to a hang of the simulation program. The log files contain data enough to reproduce the conditions that provoked the error.

The simulator also implements a mechanism to avoid falling in an infinite loop. It stores state snapshots that include the situation of the whole mapping database tree as well as all the local variables of the functions. When the system performs an iterative operation, every new state is compared to the previous ones and, if the resulting state happens to have previously occurred, the simulator ends the test, recording the malfunction in the corresponding log file. The automated tests that provoke errors can be added to the set of predefined tests.

5 Results

As a result of performing a large number of tests, many errors of different nature were discovered and thus corrected. Some errors were discovered within the very early stages of the verification process, requiring only a few iterations to rise up. However, other errors usually needed thousands of iterations before being discovered.

The final version of the mapping database has passed millions of tests, proving its robustness.

The verification process required a feedback cycle consisting of:

- Tests execution
- Error detection
- Search of the error causes
- Fix by the L4Ka Team.

The feedback cycle has a very short length since the original code must suffer very few modifications before being inserted into the simulation tool.

All the simulations and tests have been run on 32 bits x86 machines. However, due to the fact that the mapping database is virtually independent from the underlying architecture, several configurations and page sizes have been tested.

For example, one of the errors detected using the simulator was the behavior of the bit shifting operation SHL, which changes across the different generations of the x86 processors family. Originally, that operation zeroed the result when the value of the second operand was greater or equal to the size of the shifted operand. In the new generations, only the lower bits are used, leaving the first operand unmodified when the second operand equals the number of bits of the first operand.

The history of the affected files [3] can be viewed in the public CVS repository [2] of the L4Ka Team as part of L4Ka::Pistachio 0.4 [1].

6 Conclusions

The simulation tool that we have developed has proven as a powerful ally to detect and debug the mapping database subsystem of the L4 microkernel. Numerous errors were found and corrected during the whole testing process. This fact exposes the necessity of developing this kind of specific simulators to test critical subsystems.

This kind of simulators can be built for modular subsystems capable to run stand-alone. In these cases the encapsulation process can be affordable and worthy enough. The more modularity the whole system has, the easier and more effective the simulation is.

7 Acknowledgements

We must thank the whole L4Ka team and specially Espen Skoglund for their help and their patience. Without them, it would not have been possible to accomplish this project.

References:

[1] L4Ka::Pistachio microkernel main page:
`http://l4ka.org/projects/pistachio/`

[2] L4Ka::Pistachio microkernel CVSWeb:
`http://l4hq.org/cvsweb/cvsweb/pistachio/`

[3] L4Ka::Pistachio new mapping database:
`http://l4hq.org/cvsweb/cvsweb/pistachio/kernel/include/mdb.h`
`http://l4hq.org/cvsweb/cvsweb/pistachio/kernel/include/mdb_mem.h`
`http://l4hq.org/cvsweb/cvsweb/pistachio/kernel/src/generic/mdb.cc`
`http://l4hq.org/cvsweb/cvsweb/pistachio/kernel/src/generic/mdb_mem.cc`

[4] Liedtke1995. J. Liedtke, On microkernel construction, *In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15),* Copper Mountain Resort, CO, dec 1995.

[5] Liedtke1996. J. Liedtke, Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.

[6] L4Ka2005. L4Ka Team. L4 experimental kernel reference manual. October 2005.

[7] Edwards2004. S. H. Edwards, M. Sitaraman, B. W. Weide, E. Hollingsworth, Contract-checking wrappers for C++ classes, *IEEE Transactions on Software Engineering*, vol. 30-11, pp. 794-810, 2004

[8] Tuch2005. H. Tuch, G. Klein, G. Heiser, OS Verification – Now!, *In Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.