Garland, Mehta, Roehricht, Schulze

The L4 microkernel

Harrisonburg, November 29, 2003

CS-450 Section 3
Operating Systems Fall 2003
James Madison University
Harrisonburg, VA

Contents

"Our vision is a microkernel technology that can be and is used advantageously for constructing any general or customized operating system". [Liedtke et al 2001]

1 An Introduction to L4Ka

L4 is a microkernel that was created by Jochen Liedtke at IBM, and the University of Karlsruhe in the early 1990's. It is based on Eumel and L3, its predecessor, and was written entirely in assembly language for use solely with the *ix*86 processor family. Current development is done by research groups at the University of Karlsruhe[1], the University of Dresden[2] (both in Germany) and the University of New South Wales in Australia[3]. The current implementation is called L4Ka and is based mainly on research conducted over the last 20 years. The L4Ka microkernel is open source and released under the two-clause BSD license. This means that anybody is free to use the source code for his or her own purposes, whether that be for research projects, security checks, or any other use.

The L4Ka microkernel allows you to put any supported operating system on top of it. The L^{4}Linux research group at the University of Dresden is currently working on an implementation to let a Linux environment run on top of the L4 microkernel [L^{4}Linux]. Future support for the Microsoft WindowsXP operating systems is also planned [L4Windows]. In addition the famous GNU HURD operating system has recently migrated from the old Mach microkernel to L4.

1.1 About Microkernels

Microkernels are the counterpart to the well-known "monolithic kernels" presently used by most common operating systems. The name implies that the kernel is much smaller than its monolithic counterpart. Some of the more prominent microkernels are Mach (the basis for MacOS) and QNX. Whether a kernel can be considered micro as opposd to macro is determined more so on the basis of the functions it provides rather than the size of the source code. Nevertheless, we can note that the current implementation of the L4::Pistachio/ia32 microkernel has about 10,000 lines of code, compared to the Linux kernel, which exceeds 2.7 million lines.

A microkernel is a kernel that provides only basic functionality. This is typically limited to:

- process management
- memory management
- functions for synchronization and communication

All of these tasks are OS independent und run in protected kernel mode. Hardware drivers and other common operating system capabilities, such as file system management and networking protocols can be implemented in user space. These tasks are handled using a single-server or multi-servers on top of the kernel. A single-server provides all of

[1] *http://l4ka.org/*
[2] *http://os.inf.tu-dresden.de/*
[3] *http://www.cse.unsw.edu.au/ disy/L4/*

these components in one single server, as opposed to a multi-server, which uses several different servers, each doing their own task. Communication between different servers is handled using inter-process communication (IPC) mechanisms. With this design, a broken server can be restarted without having to reboot the whole machine. As the main operating system functionality is implemented in user space, fast inter-process communication is a major design goal. Old microkernel approaches tended to be slow because of poorly designed IPC mechanisms, but drastic improvements have been shown in IPC delay times with the L4

These design issues provide more scalability, extensibility, and portability when compared to monolithic kernels. Because a microkernel is "OS-neutral", different operating systems can be hosted on top of it. The system above the microkernel can be easily ported to other hardware architecture if the microkernel supports them – only the microkernel needs recompiling for the new architecture and not the OS itself or its associated applications. A microkernel can also easily be extended to work in a multi-processor environment.

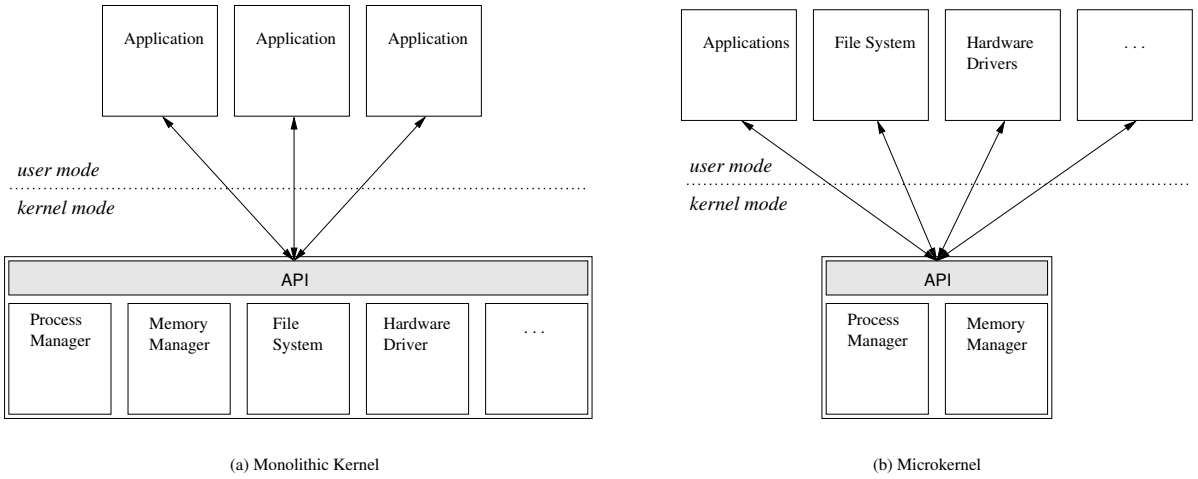

Figure 1: Comparism between a Monolithic Kernel and a Microkernel

Characteristics of microkernels:

Robustness As we mentioned before, the user of a microkernel based OS is able to restart different services which run in user space instead of having to restart the OS. This makes the system robust and highly available. Furthermore, corrupt or falsely implemented services will never harm the kernel, because everything runs in user space and has no access to the kernel memory space. Microkernels are easier to maintain and debug as their size is much smaller than the size of monolithic kernels.

Security On of the main issues nowadays is the security of a system. In monolithic kernel based operating systems, *root* has access to everything on the machine. Whenever a user makes use of the hardware he gets *root* privileges (in UNIX and Linux by *setuid root*) in a protected mode by using the drivers for the particular hardware. This

causes a potential security risk, for example, if a driver is implemented incorrectly. One famous example was the *ptrace()*-bug in some Linux kernels up to 2.4.22 which gave any user root privileges by loading a kernel module and using an exploit (more information is provided in [Szombierski 2003]). Something like that cannot happen to microkernels because the way to get root privileges is far more restricted.

Memory Usage The whole kernel (code plus data) must always be memory resident during runtime. In a monolithic kernel based OS, we cannot swap out parts of the OS that are used infrequently, but in a microkernel system we can. This idea is already commonly used even in monolithic systems (parts of the the X server in a UNIX-like operating system such as Linux can be swapped out of main memory).

Performance Whenver we will operate in kernel mode in a microkernel, interrupts are turned off to prevent critical processes from interruption. But as most services can run without making use of interrupts by the kernel this privilege can be passed better to processes with realtime requirements.

A main disadvantage of the microkernel architecture can be the communication channel. In a microkernel system that uses an operating system on top of it, communication between these two facilites must be fast because they do not have access to the same memory areas, and therefore must constantly be sending communications to each other. This is done by IPC and may reduce system performance if not handled carefully. This is a reason for the extensive research being conducted in this field. Developers of L4Ka were able to prove that costs for regular IPC calls can be reduced from typically 100ms down to under 5ms [Liedtke 1993].

2 The design of L4Ka

L4 is refered to as a "second generation" microkernel. This means the developers tried to learn from the mistakes that were made by designers of the microkernels of the first generation in the early 1990's. One of the most famous first generation microkernels was the Mach. It, along with other first generation microkernels, was not originally designed from the bottom up to be a true microkernel. The attempt was to take a monolithic kernel and source out as many services that could be outsourced (e.g. the file system or even parts of the memory management). A good microkernel should implement only basic functionality mechanisms, which can be used to achieve more complex user specific strategies in areas such as the file system and hardware management.

The developers of L4 had to decide which parts of a kernel are able to run in user space without losing functionality or security. But the L4 kernel does not need to have a concept for threads[4] or even a scheduler. The kernel only provides system calls that can handle the preemption of processes.

[4]even though L4 supports user level threads

Threads

Threads are the basic active entity in execution. Only threads can be scheduled. The communication from one thread to another is done by IPC calls. Each thread has a register set (IP, SP, user-visible registers, processor state), an associated task, address space, a page fault handler (a pager that is built as a thread and receives page faults via IPC), an exception handler, preempters and some scheduling parameters like priority or time slice.

Tasks

A task provides an environment for the execution of processes and consists of a virtual address space, ports for communication, and at least one thread. The number of threads is no longer fixed since the latest development of the L4 implementation (Pistachio 0.3). All but one thread are created inactively and can be activated via system calls (`lthread_ex_regs()`). A clan consists of one or more related tasks and every clan has a single chief task. Therefore, every task has an associated clan and chief. A task creating another task becomes the chief of the new task. A task can only be killed directly by its chief or indirectly when its chief is killed.

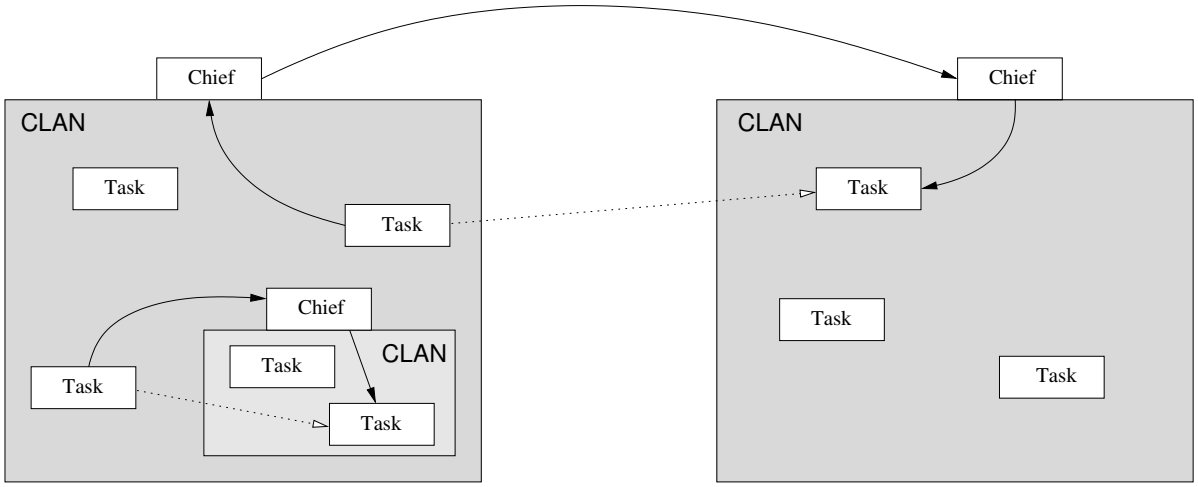

Figure 2: Tasks from different clans have to communicate via the chiefs of their clans.

Flexpages and Virtual Address Space

Flexpages are flexible large memory pages. L4 makes use of them to access main memory and I/O memory of devices. The virtual address space is made up of these flexpages, and system calls are provided to manage them:

`grant` The memory page is granted to a new user and cannot be used anymore by its former user.

`map` This implements shared memory – the memory page is passed to another task but can be used by both tasks.

`flush` The memory page that has been mapped to other users will be flushed out of their address space.

2.1 I/O Implementation

L4 does not provide native I/O support by iteself. One of the design decisions was to let I/O run in user space only. Therefore, no interface is offered to access peripheral devices directly, but the user can make use of the virtual address space functions provided by L4. The I/O address spaces of the hardware devices (including I/O memory and ports) can be mapped to the device drivers which run in user space by using `map` and `grant`. Hardware interrupts are caught by the kernel via IPC and will be passed to the device drivers. Page faults are handled the same way, which makes it possible to implement even the memory manager in user space.

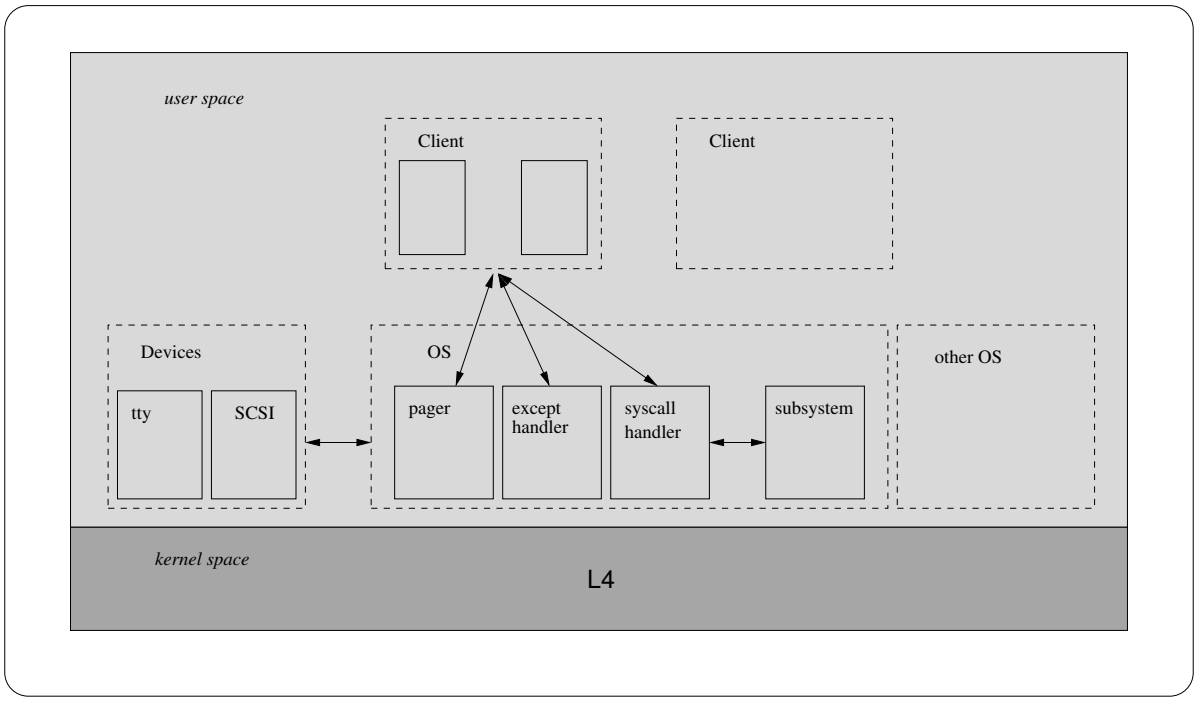

Figure 3: The Design of the Relationship between the L4 Microkernel and the Operating Systems, Devices and Applications

3 Performance

The Performance was one of the major dilemmas in the first generation of microkernels. One of the most prominent representatives was the Mach microkernel. Liedtke and other researchers evaluated the performance of microkernels compared to monolithic kernels. Most of them found that the interprocess communication is the real bottleneck in a microkernel system, because it is so heavily used. On a 486 processor, Mach needed

about 900 processor cycles for a simple system call. A system call involves switching to kernel mode and then back to user mode. To switch from one mode to the other is costly (about 80 cycles to switch to kernel mode and 20 cycles to switch to user mode on a 486 processor). Therefore, Mach's system calls caused an overhead of about 800 cycles. Liedtke's research has produced far better results [Liedtke 1993].

Some parts of the kernel are programmed in architecture-dependent assembly code, mostly those parts that are used frequently by the operating system and the applications (eg. system calls). Due to this limitation, the kernel is not hardware independant, but most parts are implemented in C++ and can be interchanged for every architecture.

The kernel provides real time applications the possibility to allocate memory that is not liable to the pager. As this part will never be swapped out of main memory, you can calculate better on its execution time.

3.1 IPC

Interprocess communication is a major part of the L4 microkernel. It had to be quite fast and also be designed carefully with regards to security issues. Even interrupts are handled by IPC calls. L4 does not use special channels to communicate, the communication is handled by threads and their *uid*s (the receiver decides if the request is granted or not on the basis of the *uid* of the incoming thread).

As indicated before, most parts of the operating system are outsourced from the kernel. This means that every device driver runs in a different address space, separate from the kernel address space. Therefore, in order to make hardware devices accessible, inter-process communication (IPC) is again necessary. Every single request to a device causes two messages (one for the request and another one for the reply). This may cause large overhead if these IPC calls are not implemented in a fair and proper manner. The main design goal must therefore be to implement a super fast IPC protocol [Liedtke et al 2000].

4 The security of the system

The security mechanism in L4's microkernel design is based on secure domains, namely, tasks, clans and chiefs. We mentioned before that the communication is not based on a channel, but is done from thread to thread. This creates a need for some control mechanisms in message passing. A clan is a group of different, but related tasks. If two threads from the same clan want to communicate with each other, it is done by normal IPC calls. Whenever two threads from separate clans want to communicate, the communication has to pass the chiefs of the clans. Those chiefs are able to either manipulate the message or simply pass it through to the other chief. A clan is assigned to only one machine and therefore if you want to pass a message over the machine's border, the protocol can change to, for example, TCP/IP.

5 Conclusion

Overall Micro kernels have been shown to be the very competitive with monolithic kernels, since micro kernels are being created with improved programming techniques. Micro kernels provide basic functionality for process management, memory management, and communication functions. Micro kernels are known to be very robust, have good security and memory usage, as well as performance. The L4 micro kernel is a second generation microkernel that is still in development stages. One of the L4's major components currently is inter-process communication and making it more efficient. Finally the main security system of the L4 is it secure domains, including tasks, clans, and chiefs.

References

[Haeberlen 2003] Haeberlen, A., Elphinsotne, K. (September 2003): User-level Management of Kernel Memory; ACSAC'03, Aizu-Wakamatsu City, Japan *http://i30www.ira.uka.de/research/documents/l4ka/userlevel-mgmt-of-kernel-mem.pdf*

[L^{4}Linux] L^{4}Linux: Linux on L4 *http://os.inf.tu-dresden.de/L4/LinuxOnL4/*

[L4Windows] WindowsXP on L4 (offered Study/Diploma Thesis) *http://i30www.ira.uka.de/teaching/thesistopics/index.php?thid=40*

[Liedtke 1993] Liedtke, J. (December 1993): Improving IPC by kernel design; 14th Symposium on Operating System Principles, Asheville, North Carolina *http://i30www.ira.uka.de/research/documents/l4ka/improving-ipc.pdf*

[Liedtke 1996a] Lietdke, J. (September 1996): Toward Real μ-kernels; Communications of the ACM, 39(9), pp. 70-77 *http://i30www.ira.uka.de/research/documents/l4ka/towards-ukernels.pdf*

[Liedtke 1996b] Liedtke, J. (October 1996): μ-Kernels Must And Can Be Small; In 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, USA *http://i30www.ira.uka.de/research/documents/l4ka/ukernels-must-be-small.pdf*

[Liedtke et al 2000] Liedtke, J. et al (September 2000): Synchronous IPC over Transparent Monitors; In 9th SIGOPS European Workshop, Kolding, Denmark *http://i30www.ira.uka.de/research/documents/l4ka/synchronous-ipc.pdf*

[Liedtke et al 2001] Liedtke, J. et al (April 2001): The L4Ka Vision; White Paper; *http://i30www.ira.uka.de/research/documents/l4ka/L4Ka.pdf*

[Pistachio] L4Ka::Pistachio microkernel *http://l4ka.org/projects/pistachio/*

[Ruckdeschel 2002] Ruckdeschel, H. (November 2002): Microkernel Betriebssysteme (Mach, L4, Hurd); Erlangen, Germany *http://www4.informatik.uni-erlangen.de/Lehre/WS02/PS_KVBK/talks/ausarbeitung-microkernel.pdf*

[Szombierski 2003] Szombierski, A. (2003): linux kmod/ptrace bug – details *http://cert.uni-stuttgart.de/archive/bugtraq/2003/03/msg00266.html*