Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel:	**Beneš Eduard**
Obor:	Informační technologie
Téma:	**Mikrojádra operačních systémů**
Kategorie:	Operační systémy

Pokyny:

1. Seznamte se obecně s různými koncepty jader operačních systémů.
2. Podrobně prostudujte koncepci mikrojader L4 a/nebo Hurd, případně i jiných.
3. Nainstalujte příslušná mikrojádra, navrhněte a proveďte vhodnou sadu experimentů k otestování jejich chování.
4. Zjištěné výsledky shrňte, diskutujte a porovnejte s teoretickými předpoklady (např. zpomalení vůči klasickým OS) a případně s výsledky různých testů dostupných na Internetu.

Literatura:

- Silberschatz, A., Galvin, P.B., Gagne, G.: Operating Systems Concepts, 6th Edition, John Wiley & Sons, 2001, 7th Edition, John Wiley & Sons, 2004.
- The L4 Mikrokernel Family. http://os.inf.tu-dresden.de/L4/
- Hurd. http://www.gnu.org/software/hurd
- The Operating Systems Resource Center. http://www.nondot.org/sabre/os/articles
- The Linux Documentation Project. http://www.tldp.org

Při obhajobě semestrální části projektu je požadováno:

- První 2 body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí:	**Vojnar Tomáš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání:	1. listopadu 2006
Datum odevzdání:	15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení:	**Eduard Beneš**
Id studenta:	84465
Bytem:	Čajkovského 7, 949 11 Klokočina
Narozen:	07. 05. 1983, Nitra

(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP:	Mikrojádra operačních systémů
Vedoucí/školitel VŠKP:	Vojnar Tomáš, Ing., Ph.D.
Ústav:	Ústav inteligentních systémů
Datum obhajoby VŠKP:	....

VŠKP odevzdal autor nabyvateli v:

tištěné formě	počet exemplářů: 1
elektronické formě	počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - [✓] ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....

....	[illegible]
Nabyvatel	Autor

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MICROKERNEL BASED OPERATING SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE EDUARD BENEŠ
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MIKROJÁDRA OPERAČNÍCH SYSTÉMŮ
MICROKERNEL BASED OPERATING SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE EDUARD BENEŠ
AUTHOR

VEDOUCÍ PRÁCE Ing. TOMÁŠ VOJNAR, Ph.D.
SUPERVISOR

BRNO 2007

Abstrakt

Táto práca sa zaoberá problematikou mikrojadier operačných systémov. Prvá časť je zameraná na oboznámenie s problematikou jadier operačných systémov. Obsahuje základné vlastnosti a mechanizmy druhej generácie mikrojadier reprezentovanej mikrojadrom L4, na ktoré sa zameriavame v ďalších častiach práce. Následne sú opísané dva rôzne porty operačného systému Linux nad mikrojadro L4, sú to L4Linux a Wombat. V druhej časti práce je popísaný spôsob inštalácie vybraných portov a hlavné problémy, ktoré sme museli riešiť. Tretia a štvrtá časť sú zamerané na problematiku testovania výkonnosti nainštalovaných systémov. Popisujeme metodológiu zvolených experimentov a význam jednotlivých testov. Výsledky, spolu s ich vyhodnotením, sú uvedené vo štvrtej časti. Pokiaľ to je vhodné, získané výsledky konfrontujeme medzi sebou, prípadne s výsledkami testov získaných z Internetu. V záverečnej časti je na základe nadobudnutých znalostí uvedená stručná diskusia na tému možností uplatnenia mikrojadier.

Klíčová slova

mikrojadro, monolitické jadro, Linux, L4, Wombat, L4Linux, výkonnosť

Abstract

This thesis discusses the area of microkernel based operating systems. The first part is focused on the familiarization with the issue of the operating system kernel. This part contains the basic characteristics and mechanisms of the second generation microkernels represented by the microkernel L4, on which there is major focus in other parts of the thesis. Subsequently, there are described two different ports of the operating system Linux on top of L4 microkernel and those are L4Linux and Wombat. In the second part of the thesis, the method of the installation of the given ports is depicted and the main problems the author had to face. The third and forth part are focused on the issue of testing the performance of the installed systems. The methodology of chosen experiments is described and the meaning of the individual tests explained. The results, as well as their evaluation are stated in the forth part. If suitable, the gained results are compared with each other, or with the results gained from the Internet. In the last part, a short discussion is conducted based on the gained knowledge on the issue of the possibilities of microkernel application.

Keywords

microkernel, monolithic kernel, Linux, L4, Wombat, L4Linux, performance

Citace

Eduard Beneš: Microkernel based operating systems, bakalářská práce, Brno, FIT VUT v Brně, 2007

Microkernel based operating systems

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáša Vojnara, Ph.D.

. . . .
Eduard Beneš
May 10, 2007

Poděkování

I would like to thank Ing. Tomáš Vojnar, Ph.D. for his guidance, very helpful comments, and advices. I also like to thank everyone who has proofread this document for their useful feedback and recommendations. Finally, many thanks go to my family for giving me the possibility to study at the university and their constant support.

Contents

Chapter 1

Introduction

In this work we present experiments on performance of L4 microkernel based operating systems. First generation of microkernels represented by Mach microkernel evolved from earlier monolithic kernel approaches and they were inefficient and inflexible [25], while second generation microkernels like L4 rigorously aim at minimality and are designed from scratch [14]. We had installed selected Linux ports on top of L4 microkernel and repeated several earlier experiments and conducted several new ones.

In the next chapters we present kernel and microkernel characteristics, then we focus on L4, second generation microkernel, and various design concepts of ports of Linux operating system on top of L4 microkernel. Specifically we focus on L4Linux and Wombat and a description of installation process and associated problems follows. Later we present, compare and evaluate performance results for micro and macrobenchmarks executed on installed systems to get an idea on performance of microkernel based systems.

1.1 Overview

Original assignment for our work was:

- Become acquainted with different kernel concepts and operating systems.
- Perform detailed research of L4 and/or GNU/Hurd microkernel concepts.
- Install chosen microkernels, propose and perform appropriate set of experiments to test their behavior.
- Discuss and evaluate gathered results and compare with theoretical expectations (i.e., slowdown when compared to monolithic OS) and results available on Internet.

To achieve the goals of our work the project moved through different phases. In the next subsections we will briefly describe outline of our work.

1.1.1 Research On Topic

Purpose of the first phase was to get introduction to the problems of microkernel based operating systems and gain necessary theoretical knowledge needed for successful evaluation and resolution of problems that will come up in upcoming phases. In this phase we have subsequently conducted research for material related to problematic of microkernel performance and operating systems based on L4 or Mach microkernel that could be used for the final performance testing and will make it possible to set and achieve our goals. During this familiarization process we have decided to focus on second generation of microkernels represented by L4 microkernel. There were several reasons that supported this decision. First generation of microkernels had been proven slow and unsatisfactory many times before and thus led to the bad reputation of microkernels. This has been broken by second generation of microkernels under guidance of Jochen Liedtke, the father of L4 microkernel, which is around for more than decade and during this time went through many changes.

In the past, many performance tests were done with the purpose to disclose deficiencies of the first generation and help to learn from the mistakes done in design and implementation of the first generation of microkenels based mainly on Mach microkernel or Minix. We decided to find some new, freely available (open-sourced), implementations of systems less or more compatible with Linux and based on some member or L4 microkernel family. After some research we have focused on Linux ports on top of L4 microkernel: L4Linux, being developed at Technical University Dresden, and Wombat from NICTA at UNSW, Sydney Australia.

As an alternative goal was set GNU/Hurd that uses Mach microkernel and represents the first generation of microkernels. This system does not provide the performance and stability that would be expected from a production system and only about every second Debian package has been ported to GNU/Hurd [1]. The GNU/Hurd is based on Mach microkernel which does not strictly follow the minimal microkernel design and due to its slow performance some drivers reside in the kernel [2]. We installed Debian GNU/Hurd operating system, but it will not be discussed in this paper any more.

1.1.2 Research into L4Linux and Wombat

This phase was relatively straightforward and consisted of research on L4Linux and Wombat, gathering theoretical information, capabilities of final systems and necessary installation requirements. Lack of documentation, installation instructions, or outdated information played the main role of this phase. About the same time we started primary installation phase of selected Linux ports.

1.1.3 Installation

Installation of the L4Linux and Wombat took much more time than it had been expected. During this phase we gained knowledge about many parts of Linux kernel, virtual machines like VMware, QEMU and many other topical problems. First we started with installation of a basic environment required for latter installation of Linux ports on top of L4 microkernel. We have found out that installation of microkernel based on real hardware can be much more complicated than running it within a virtual machine. Another essential problem was lack of documentation or outdated information. At the end we had two running Linux ports on top of L4 microkernel.

1.1.4 Performance Testing

The proposed set of experiments was performed after successful installation of both microkernel based systems on native hardware to test their behavior. We planned to run the same set of tests on L4Linux and Wombat. This showed up to be impossible due development phase of Wombat. Many macrobenchmark tests done in the past present that average slowdown of a L4Linux system ranges from 5% to 10% [14]. This has been proven true only partially. Running tests that did work on large files disclosed that the performance dramatically declines. This has been proven by evaluation of our microbenchmark results.

1.2 Related Work

During the preparation phase we had to read through many papers that discussed performance of microkernels. This helped in our decision to focus on second generation of microkernels specifically L4 microkernel. The first generation of microkernels represented by Mach has poor performance and this has been only partially overcome by placing some drivers back into kernel breaking the minimal concept proposed by Jochen Liedtke that we would like to follow.

In this paper we focused on performance of systems based on L4 microkernel. There were written many papers which discussed this topic. The most important paper on performance of L4 microkernel was the paper by Liedtke et. al. for us [14]. This paper discusses performance of L4Linux and compares it to MkLinux and native Linux. Another work on performance of L4 has been done by Herder et. al. [17] to explore the performance of a system with drivers running in user space. Adam Lackorzynski proposes L4Linux porting optimizations in [20]. Experimental study on the performance of L4Linux on top of L4 is presented by Guanghui in [8]. Performance of Wombat system is discussed in paper by Leslie et. al. in [6].

Chapter 2

Microkernel Based Systems

With the growing popularity of UNIX operating system, the kernel started to grow in size and became difficult to manage. This led to development of first generation of microkernel represented by Mach and carried out by researchers at Carnegie Mellon University.

First are presented different approaches in kernel and system design. After short comparison of microkernel and monolithic kernel follows the description of various microkernel concepts and finally introduction to the Linux ports on top of L4 microkenel. During this work we had to deal with lack of documentation or other reliable information sources. Finding reliable information was difficult. Easily available documentation is outdated, incomplete or discusses only too specific topics not related to our work.

2.1 Different Kernel Approaches

There are many different definitions of the operating system kernel depending on the point of view. Operating system kernel is the part of the system which executes in the privileged mode of the underlying hardware. A more common definition is that the kernel is the one program running at all times on the computer, with everything else being application programs [2].

During the evolution different approaches to the construction of kernel appeared. They usually differ in the set of services running in kernel mode, primitives and level of provided abstractions. The following are the main types while there exist many different designs which combine these approaches into so called hybrid kernels. It should be noted that the definitions may slightly vary from source to source and even the main set of different types depends on the point of view.

Monolithic kernel All services are running in kernel space in supervisor mode. These kernels usually define a high-level abstraction layer over computer hardware with large set of primitives and many system calls. Modern monolithic kernels support so called loadable modules, which can be dynamically loaded or unloaded from the kernel at runtime. Typical examples are: Linux, Unix kernels, Microsoft Windows 9X series, OpenVMS, MS-DOS.

Microkernel Microkernels provide only basic services running in kernel mode. Other services are provided by so called servers which are user-space programs outside the kernel. They usually have a small number of system calls and provide only a small set of primitives. Typical examples are: L4, Minix, QNX, Mach.

Nanokernel This type of kernel is even more minimalistic than the microkernel and represents the closest hardware abstraction layer of the operating system. It is commonly used to host other operating systems as a hardware abstraction layer to increase portability or in real-time systems. Typical examples are: KeyKOS, AdeOS.

Exokernel This is kernel developed at MIT. It tends to force as few abstractions as possible, enabling them to make as many decisions as possible about hardware abstractions. Their functionality is limited to ensure protection and multiplexing of resources. Thus the kernel provides and allocates the physical resources and the programs can decide what to do with these resources. Typical examples are: Aegis, XOK.

2.2 Comparison of Monolithic Kernel and Microkernel

The two most often discussed kernel designs are monolithic kernels and microkernels. Whether one or the other design is better was a topic of debate between Andrew Tanenbaum and Linus Benedict Torvalds fifteen years ago and continued in the year 2006 as Tanenbaum-Torvalds Debate, Part II [5]. In this section we present the main differences, advantages and disadvantages of each design, rather than deciding which design is better.

As it has been already mentioned, the main difference between microkernel and monolithic kernel is in the number of services and policies provided by the kernel. Monolithic kernels provide many services and policies as shown at the left side on Figure 2.1. The kernel contains the entire operating system linked in a single address space and running in kernel mode. It may be structured but there are no protection boundaries around the components [17]. A bug in a device driver might crash the entire system. The main advantage of monolithic kernels is their speed, simplicity of design and huge community around it. As summarized in [17] the problems of monolithic kernels resulting from their design are:

- No proper isolation of faults.
- All code runs at the highest privilege level.
- Huge amount of code implying many bugs.
- Untrusted, third-party code in the kernel.
- Hard to maintain due to complexity.

Microkernel based systems have only tiny kernel providing basic services e.g. IPC, scheduling, interrupt handlers, process management, but ideally nothing else. File system management, device drivers, and so on is provided as unprivileged user-mode servers. They are running in separate address spaces isolated from the others. This model can be characterized as a *multiserver operating system* [17] and is depicted on the right part of Figure 2.1 from [27].

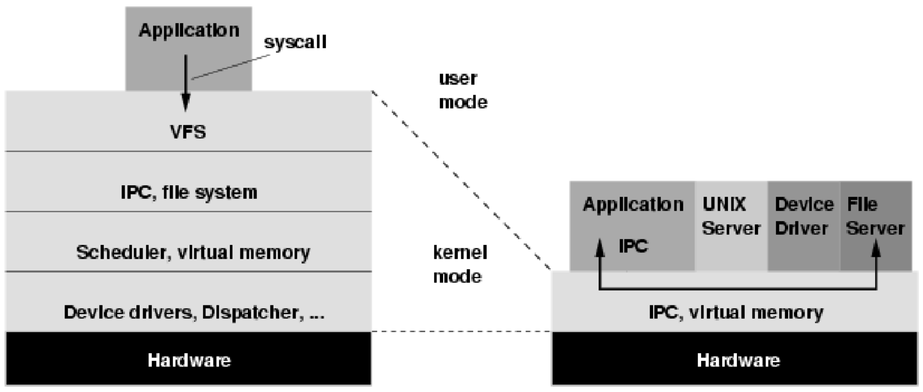


Figure 2.1: Microkernel vs. monolithic kernel

Advantages of microkernel design presented by Liedtke in [25] are:

- Simplicity.
- Flexibility and extensibility.
- All servers can use mechanisms provided by microkernel.
- Modularity because of its clean interface.
- Adding a new services without kernel modification.
- Small Trusted Computing Base.
- Easier maintenance since the kernel is smaller.

In the next few sections is the microkernel design described more in detail.

2.3 Microkernel

Microkernel structures the operating system by providing only basic operating system services in the kernel and moving all other services to userspace. Basic services include process management, memory management, and functions for synchronization and communication.

This approach promised dramatic increase in flexibility, safety and modularity [25]. Microkernels evolved since late 1980's through many academic projects and disappointments into second generation of microkernels led by Jochen Liedtke and today represented by L4 microkernel or commercial QNX microkernel. The right side of Figure 2.1 shows structure of typical microkernel based system.

2.4 L4 Microkernel

Many years of research and development towards the performance enhancement of first-generation microkernels did not bring awaited results when compared to performance offered by traditional monolithic kernels. This caused microkernel to gain reputation of being too slow and lacking sufficient flexibility [14] and were put aside by many kernel developers. The drawbacks of the first generation of microkernels led to reexamination of used concepts. This made Jochen Liedtke believe that microkernels should be as small as possible and that proper design can bring performance, flexibility, and correctness [24].

The minimal concept introduced by Jochen Liedtke [25] implements in the kernel only address spaces, interprocess communication, and basic scheduling, stating: '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' [23].

All other functions of standart operating system (e.g. pager, device drivers, networking, file system, etc.) that are present inside monolithic kernel run in user mode resulting in more secure and reliable system. Running in user-mode does not mean running the entire system in single user mode server that would bring the same problems as with monolithic kernel. Each untrusted module is running as a separate user mode process that is isolated from others. Thus buggy driver can not crash the entire system. A detailed discussion on this topic can be found in [17].

2.4.1 Overview of L4 Kernels and Environments

Before we can take a closer look at L4 microkernel we should familiarize ourself with different kernel versions and implementations, environments and operating systems based on L4 microkernel. The following information were gathered from `l4hq.org`.

During the decade of L4 development, the kernels API and ABI went through some changes. To help reader better orientation while reading this work, we will shortly describe currently available implementations of L4 microkernel.

OKL4 OKL4 is commercially developed and supported successor of NICTA::Pistachio-embedded. This newest kernel implementation of L4 features many improvements and optimizations, is scalable from embedded to large multi-processor systems. The kernel API used is OKL4 v2 and supports ARM, IA32, AMD64/x86-64, Mips32/Mips64 architectures.

L4Ka::Pistachio Pistachio was developed by researchers at the University of Karlsruhe in collaboration with University of New South Wales, Australia, short UNSW. It implements L4 Version 4 kernel API, code-name Version X.2. It supports IA32, IA64, AMD64, ARM, PowerPC-32, PowerPC-64, Alpha and Mips64 architectures.

Fiasco Fiasco is designed as a preemptive real-time kernel. It supports IA32 and ARM architectures. It has Version 2 kernel API, the L4 API originally implemented by Liedtke, and also supports Version X.0 API which is an experimental successor of Version 2.

NICTA::Pistachio-embedded As the name says, this is descendant of Pistachio developed by NICTA group at UNSW optimized for embedded systems. It supports IA32, ARM and Mips64 architectures. Development of this implementation has been discontinued.

L4Ka::Hazelnut This is implementation of Version X.0 API almost completely written in C++ with intention to be portable across IA32 platforms. Besides IA32 it supports ARM architecture. Development of this implementation has been discontinued.

L4/Alpha, L4/MIPS, L4/PowerPC These are L4 implementations for Alpha, MIPS and PowerPC platforms based on Version 2 kernel API. They were developed by NICTA group at UNSW and the development has been discontinued.

The L4 microkernel is really minimalistic. Therefore to simplify development and unify development effort Kenge and L4Env programming environments were developed, providing basic services like memory management, less or more complete libc functionality, etc.

2.4.2 L4 Fundamentals

In this section are described some basic abstractions and mechanisms important to understand when talking about L4 microkernel. These concepts may vary in some implementation details for different L4 kernel API Versions. More detailed information can be found in [19], [18], [12], and [15] that were used as an information source.

Gernot Heiser summarizes in L4 User manual [12] Liedtke's opinion on fundamental abstractions that should be provided by microkernel:

- **address spaces** because they are the basis of protection,
- **threads** because there need to be an abstraction of program execution,
- **inter-process communication** (IPC) as there needs to be a way to transfer data between address spaces,
- **unique identifiers** (UIDs) for context-free addressing in IPC operations.

Threads

Threads in L4 are the basic active entity. They execute within an address space and can only access memory from the address space in which they execute. Each address space can have multiple threads. Only threads can be scheduled. The communication from one thread to another is possible using shared memory or IPC facilities and will be described later on. Each thread has a register set (IP, SP, user-visible registers, processor state), associated task, address space, a page fault handler (separate thread called pager), an exception handler, preempting and scheduling parameters.

Tasks

A task provides an environment for the process execution and consists of a virtual address space and at least one thread. The number of threads in task is no longer fixed since the latest development version of the L4 implementation (kernel API Version 2 has fixed number of tasks).

Address Spaces

Address spaces in L4 are constructed recursively. The initial address space is sigma0 and represents physical memory. At the start all other address spaces are empty. Construction and management of other address spaces is done by three basic operations on virtual pages grant, map and flush [23].

- **grant** can be done only by the owner of an address space that he wants to *grant* to another space, provided the recipient agrees. This removes the page from the granter's address space and including it in grantee's address space.
- **map** operation maps region from caller's address space into another address space, provided the recipient agrees. Resulting in situation when the region is being accessible in both address spaces.
- **flush** also referred to as unmap operation allows owner of an address space to flush any of its pages. The pages are unmapped from all other address spaces which directly or indirectly received them.

Grant and map operations induce a tree data structure, which contains the physical frames as root node, address spaces as nodes which are connected due to grants and maps for the address spaces. There is a tree for every physical frame, called tile-map tree [16].

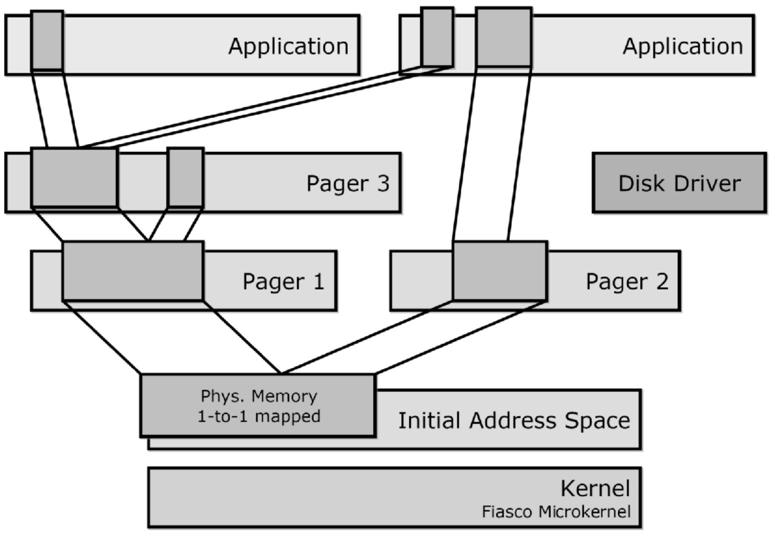


Figure 2.2: Message redirection by Clans&Chief

Since grant,map, unmap operations are done on virtual pages, they can only affect pages that are mapped into the caller's address space. They are sent as messages and have to be explicitly received by target thread. Controlling I/O rights and device drivers is done by memory managers and pagers on to of the microkernel [23].

Pagers are special threads designed to handle page fault notification messages by executing appropriate operations described above. These memory concepts allow memory management to be implemented outside the microkernel as user level servers [23]. Figure 2.2 from [33] is example of tree-like structure with hierarchical pagers.

Interprocess Communication

For a microkernel system where services are implemented at user level, communication between those two environments must be very fast. They do not have access to the same memory areas thus they must constantly be sending communication to each other. This communication is done by IPC and may reduce the overall system performance as it can be seen in some first generation of microkernels.

Interprocess communication on L4 is message based and is the most important part of L4 microkernel. One of the requirements for its features was to be extremely fast and carefully designed with regards to security issues. L4 has synchronous IPC with no need to buffer or copy data. Synchronous IPC requires an agreement between both the sender and the receiver. Threads have message registers for passing IPC messages. Sending thread writes its message into its message registers and receiving thread reads message from its message registers. If sender or receiver is not ready the other party must wait. Unbuffered IPC reduces the amount of copying involved and plays significant role in high IPC performance. This is the basic concept for all types of operations and messages.

The L4 microkernel provides only seven system calls in Version V2 and eleven in Version X2 kernel API. Everything else must be build as user level server, so IPC is the only means of communication between them. This means that it is used to pass data by value or by reference, for synchronization, wake-up calls, pager invocation, exception handling, interrupt handling. Even device control is registered via IPC [12].

Clans & Chiefs

Clans&Chiefs is the security policy model has not been used since kernel API Version 2 and is not being used since kernel API Version X.0 and Version X.2. This concept is one of the security mechanisms used by L4. It allows implementation of protection policies using IPC message transfers. It has been found to be inefficient, inflexible, and prevents dynamically changing security policies to be implemented efficiently [18]. Security policies are important because they allow controlling IPC and the information flow.

In this model, a clan is composed of all the tasks created by the same one task. A task creating another task becomes the chief of the new task this means that every clan has only one chief. Chiefs can enforce different security policies [32]. A task can be killed directly only by its chief or indirectly when its chief is killed.

Threads can directly send IPC only to other threads in the same clan, or to their chief. If a message is sent to a member of different clan, that message is instead delivered to the chief of sender's clan, who may or may not forward the message. Clans may be nested. If a message is sent to a member of a subclan of the clan containing the sender, that message is delivered to the task in the clan whose clan contains the addressee [12].

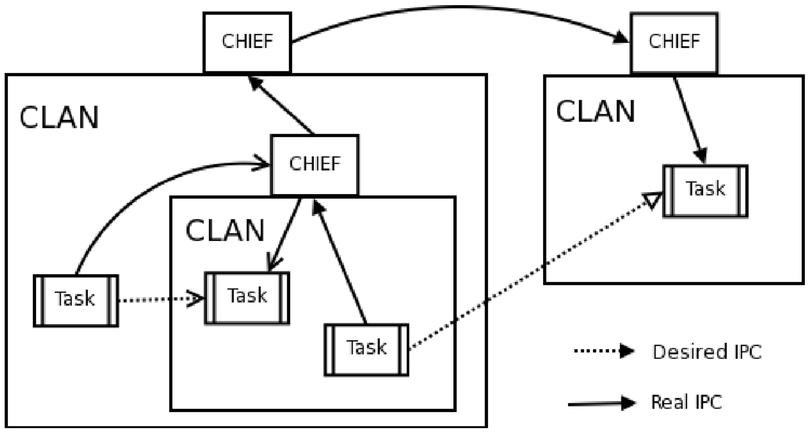


Figure 2.3: Message redirection by Clans&Chief

IPC Redirection

This is the new security model implemented in IBM internal versions of L4 using kernel API version X.0. This mechanism makes two improvements: (1) it removes the management overhead of redirection policy from the kernel, so access control enforcement can be implemented outside the kernel, and (2) it separates the notion of who controls the redirection policy from the redirections themselves, so redirections can be configured arbitrarily and dynamically [32]. While the first improvement has been met in Clans&Chief model, the later was not. It should be mentioned that in this model it is possible to implement Clans&Chiefs on to of IPC redirection with the proper redirectors.

Each thread has an associated redirector or also called redirection-controller. Redirector is another thread of the same user-level task, or more often another user-level task. Its purpose is to controll incoming, outgoing IPCs or both. Redirection controller is privileged to set redirection policy for the threads in its redirection set. Since a redirector is user-level thread the system designers are free to implement redirection policy as desired for their system [32].

Redirectors can be changed at run-time, and can be stacked, setting a redirector to a thread already acting as a redirector. This was not possible in the Clans&Chiefs model. As it has been already mentioned, the Clans&Chiefs was too complex and too slow in some cases, even if a single IPC is fast, and this can be easily reduced by dynamicall configuration.

The IPC redirection mechanism can be very useful, for example to monitor applications or to allow running untrusted code inside so called sandbox preventing the code to interact directly with the operating system. Even untrusted binary code could be run directly on main CPU without any need of virtual machine, and without any risk from the security pint of view [11].

Scheduling

The L4 microkernel was designed to allow implementation of user-level scheduling. [12]. This means that the L4 provides in the kernel only basic scheduling capabilities and implements simple internal scheduler. Thread scheduling can be controlled using timeslice length, thread priority and maximum controlled priority. Each L4 thread has a timeslice length associated with it and will be scheduled for the timeslice associated with it. After the time quantum expires, the next runnable thread will be scheduled. It should be mentioned that timeslice length is not determined by its priority. The kernel defines 256 levels with 255 being the highest priority. Usually used for user level schedulers or interrupt handlers. As it has been already mentioned, L4 microkernel has an internal scheduler. It is based on hard priorities using round-robin within priority levels. Each priority level has its own queue, empty if there are no threads with such priority. Kernel manages ready list per priority in its ready queue. If thread's priority is changed the queue it belongs to will change.

Interrupts

Each interrupt can be registered to an interrupt handler represented by a user-level thread. Hardware interrupts are caught by L4 microkernel. Upon each reception of interrupt by the kernel, it is delivered via IPC to the corresponding interrupt handler. Even the device drivers are in correspondence to L4 microkernel philosophy implemented as user level servers. This would not be possible without described interrupt mechanism.

Exception Handling

Another interesting mechanism used in L4 is the exception handling. Threads can have specified exception and a system trap handlers. When exception or system trap is made by user thread, it is converted by L4 into IPC message and forwarded to the thread's exception or system trap handler. With the IPC message is being sent the state of thread and registers. The handler resumes the thread by sending a reply message containing the modified state.

2.5 L4Linux

In the previous sections the reader became familiar in general with some basic concepts of L4 microkernels. This section describes Linux port on top of L4 microkernel called L4Linux which provides full binary compatibility with the original Linux kernel. This means that both Linux kernel and Linux applications run unmodified on L4Linux as L4 user level tasks. The performance of this Linux port when compared to monolithic Linux is presented as comparable or small [14]. Whether this is true is the subject of this work and is evaluated in the next chapters. Information presented in this section are mainly based on Adam Lackorzynski's document 'L4Linux Porting Optimizations' and lectures provided at TU Dresden on L4 microkernels.

The first implementation of L4Linux is more than 10 years old and was carried out by the Operating Systems Group in Dresden in 1996 and based on Linux 2.0. One of the important milestones was the support for X.2 API as well as symmetric multiprocessors (SMP) introduced in L4Linux-2.4. SMP is supported by L4Ka::Pistachio and L4Linux-2.4. The latest release of L4Linux at the end of writing this paper is based on Linux 2.6.20 and is under continuous development.

2.5.1 System Architecture

The L4Linux is a key component of Dresden Real-Time Operating System project, for short DROPS, which is a research project aiming at the support of applications with Quality of Service requirements [34]. The goal of this project is to build a system with real-time and non-real-time applications running side by side. At the start of the development of L4Linux, no real-time guarantees were possible in Linux, it is located in the non-realtime part of DROPS where it provides an environment to run unmodified Linux applications. Figure 2.4 show the architecture of DROPS system as it is presented in [33].

Underlying system services are used by L4Linux in the same way as by any other L4 applications. In other words, the L4Linux can use unmodified Linux drivers. Another approach is to use so called stub drivers together with external L4 servers providing desired functionality. This is, for example, necessary for virtualization when running multiple L4Linux instances is desired, because hardware can be used only by one instance at a time. As an example network server could be used. L4Linux can use services provided by network server capable of sharing resources among realtime and non-realtime clients.

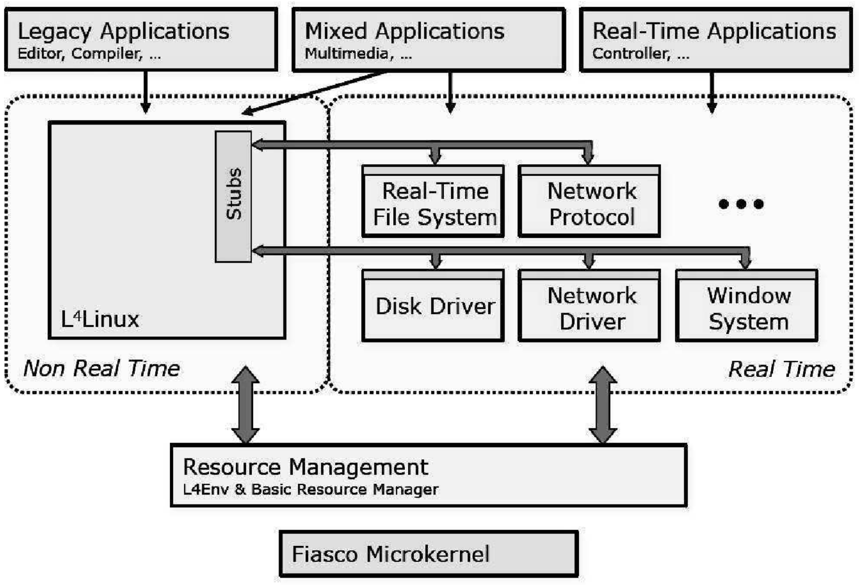


Figure 2.4: DROPS system architecture

L4Env

The L4 microkernel API is small and provides only minimal functionality to user level applications. This makes it hard to use without supporting libraries and services. With the growing number of L4 applications an unified environment began to be needed. Combining components developed by different authors became problematic since they used their own libraries even for common functions like `printf`.

L4Env is a programming environment for application development on top of the L4 microkernel family and is being developed as part of the DROPS. Its purpose is to define a minimal environment that would be available for every L4 application including a C library, memory management, thread management, program loading, event handling, interface definition language (IDL), stub generator and others. Its function is to abstract L4 concepts to higher levels and thus provide the needed libraries and services as servers in user space.

The most important servers [7] that will be build during the installation process of L4Linux are:

bootstrap Bootstrap for L4. It is responsible for initial setup of regions and the kernel page.

main Actual Fiasco L4 microkernel.

sigma0 System's pager that possesses all pages at start-up. Servers can have their own pagers.

roottask Basic resource manager. It is responsible for physical memory, interrupts, tasks, small address spaces, and launch of basic servers.

events Communication server that provides communication between L4 tasks.

names Basic namespace manager, responsible for registering and unregistering names with a thread_id, etc.

log Basic console and support for logging.

dm_phys Physical dataspace manager for L4.

simple_ts L4's simple task manager which provides task initialization, task management, task queue, and timeout management.

rtc Real Time Clock server.

l4io Multiserver providing an interrupt manager omega0 and basic I/O management.

l4exec Binary interpreter for L4 interpreting L4 binaries and creating dataspaces which contain program sections. It is also capable of resolving dependencies with library modules. Currently supports ELF only.

bmodfs Minimum boot time file provider.

loader Dynamic L4 loader capable of loading binaries at runtime. Its responsibility is to fetch files from the file provider and hand them to l4exec.

2.5.2 Implementation

When porting Linux on top of L4 Linux, it was decided to reuse the Linux source code with minimal modifications. This has been kept. The L4Linux defines new architecture which uses L4 and L4Env functionality and makes it possible to reuse nearly all parts without modifications from native architecture. This was possible because the L4 kernel encapsulates the differences of the hardware and provides a small but very powerfull interface [31]. Structure of L4Linux version 2.6 is illustrated in Figure 2.5 from [33].

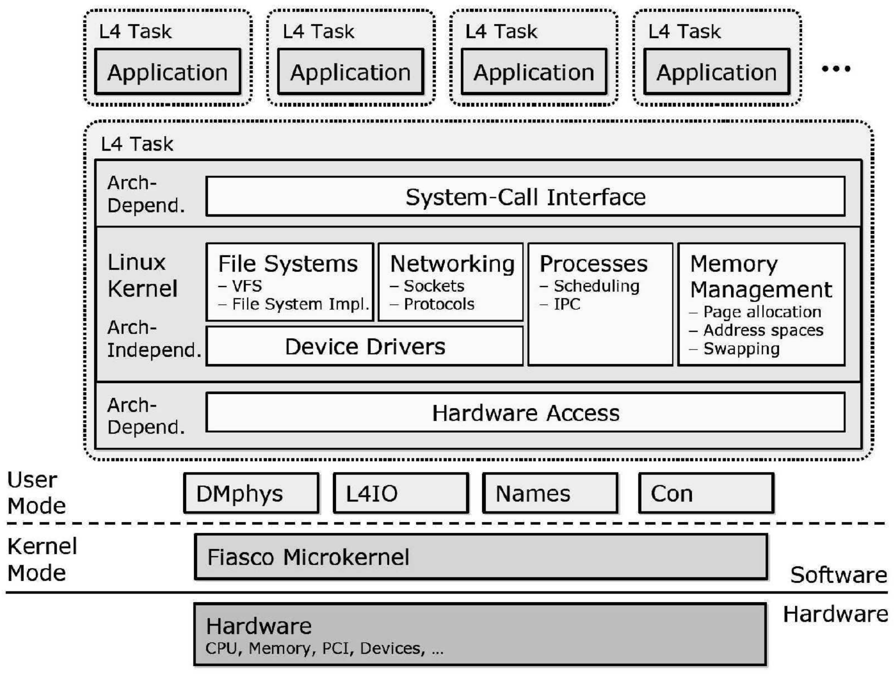


Figure 2.5: L4Linux structure

Linux kernel and Linux user processes run each within separate L4 tasks. There is separate task for each user process. Architecture dependent part implements required functionality using L4 and L4Env primitives while most parts are used unmodified from native, e.g. x86, architecture.

The thread that executes the Linux kernel code is called Linux server. In the running system it is in idle loop waiting to handle system calls, exceptions, and page faults of user processes. The server task uses one or more threads to handle interrupts depending on the L4 version.

Page faults of the Linux server are handled by a thread called the root pager. Linux server is a pager for all user processes, page faults are sent to the L4Linux server. When user process page fault occurres, L4Linux has to map memory into the address space of the requesting process using the L4 map and unmap operations. All memory received by user processes comes from the Linux server which manages all the memory from the L4Linux system. For remapping memory from the Linux server to the address space of the Linux server again, the Ping-Pong task is used.

Another interesting design concept was used for signal handling. In L4Linux version 2.4.x there are two threads within the task that handles user process. One of them executes the user program, while the other, so called signal thread waits for commands from Linux server to manipulate the user thread.

System Calls

In native Linux, if application needs to carry out system call, it enters the Linux kernel using the interrupt gate 0x80 by executing `int 0x80` instruction. This is not possbile on L4 systems where the corresponding instruction will cause an exception and the causing program will be terminated without installed exception handler.

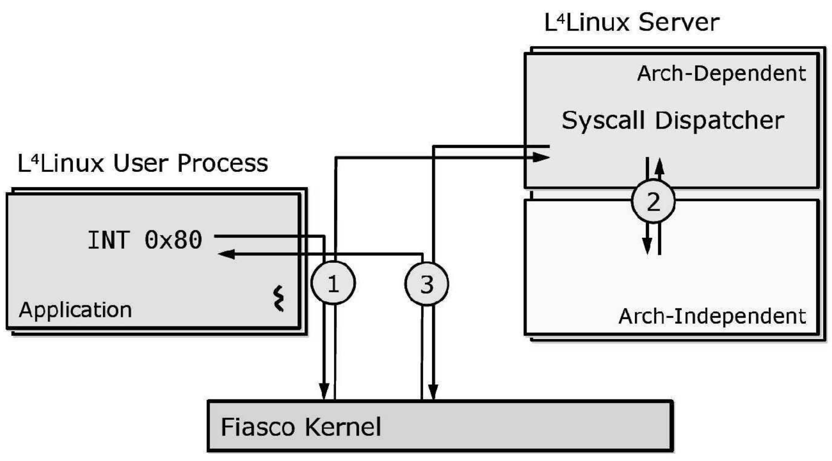


Figure 2.6: System call mechanism in L4Linux

In L4 an application executing `int 0x80` instruction triggers an exception to the L4 kernel. Then an exception IPC is sent to the L4Linux server by the L4 kernel. When L4Linux server receives exception IPC, it handles the system call and sends an exception reply to the L4 kernel. Upon reception of reply, new state of thread is set. Looking closer at this mechanism the cost of this operation can be seen. While in native Linux system call costs 1 kernel entry and no address space switch, in L4Linux it costs 2 kernel entries and exits (exception IPC and reply) and 2 address space switches (user task - kernel task - user task). This situation is shown in Figure 2.6 from [33].

Scheduling

There are two active schedulers in L4Linux system:

- L4 kernel
- Linux server

The L4 kernel is scheduling all L4 threads including L4Linux processes. It should be noted that L4 scheduler uses static priorities and schedules threads with the same priority round-robin. Linux server has the scheduler for Linux processes. From the point of view of the Linux server, multiple user processes can be running in the L4 system. Linux server needs to distinguish between the user processes being served by the Linux server and the processes running at the moment, because the Linux scheduler can only consider such processes that are blocked by the server.

Interrupt Handling

In the L4 microkernel the interrupts are translated into synchronous IPC message and sent to the thread attached to the interrupt. In L4Linux the interrupt messages are received in separate threads either directly from L4 kernel or from L4IO through the omega0 interface. Interrupt threads run on higher priority than the main thread and the threads of user processes. L4Linux emulates prioritization of hardware interrupts by giving interrupt threads different priorities.

2.6 Wombat

As a second Linux port on top of L4 microkernel we chose Wombat which is a complete paravirtualized Linux kernel running on top of Iguana. Paravirtualization means that L4 is used as a new architecture that Linux can run on. There is a similar approach used in L4Linux. Wombat is presented as highly portable and currently runs on x86, ARM and MIPS architecture.

This Linux port and the resulting system is in many design features similar to L4Linux, which pioneered an idea to run Linux in user mode on top of the very small and fast microkernel. Sarma and McKenny presented this as a third approach to real-time computing with Linux in [9]. Figure 2.7 from [6] shows the resulting system.

The L4 microkernel is designed to satisfy the needs of hard realtime systems and is responsible for dealing with interrupts. The Linux system is not involved in any of the interrupts destined for realtime tasks. The system is divided on the Figure 2.7 by black line into two isolated parts. The untrusted part where Wombat Linux server resides and the sensitive realtime part. Crash of Linux or other untrusted application cannot compromise the real-time side, and if needed the Linux can be easily restarted without requiring any system downtime. Another advantage of this approach is small Trusted Computing Base (TCB) because it only contains L4 microkernel and Iguana Resource Manager that have less than 30 000 lines of code together.

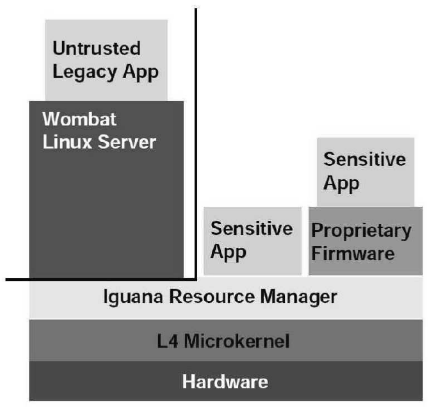


Figure 2.7: Running Wombat and other applications on a microkernel

The next few pages are based on available information gathered from webpages of National ICT Australia, available documentation on the discussed topics and [6].

Wombat has been chosen for several reasons:

- The microkernel and Iguana are open-sourced under BSD license. Linux runs on top of it, that is why firmware and other applications do not need to be under GPL.
- Embedded systems are the biggest potential market for Linux [6], while Wombat can provide needed legacy support for embedded devices.
- The developers present that Wombat's performance benchmarks show that it performs very well when compared to native Linux, and even outperforms it in some benchmarks.
- For building applications on top of L4 it uses different environment than L4Env.

2.6.1 System Architecture

System structure with Wombat on top of L4 microkernel with Iguana is shown in Figure 2.8 as it is presented in [6]. Iguana address space is represented by a light gray area with separated protection domains. Inside the shared address space runs Wombat in its own protection domain. Running in the shared address space simplifies sharing and allows fast context switching. The system's minimal TCB is composed of basic OS services provided by MyOS servers and device drivers. Application codes that depend only on OS services and not on the presence of Linux are called Iguana User Processes. This could be e.g. firmware or other sensitive applications as pictured on Figure 2.8. In some cases programs may need their own address space. This is theoretically possible, but they would partly or completely lose access to Iguana services.

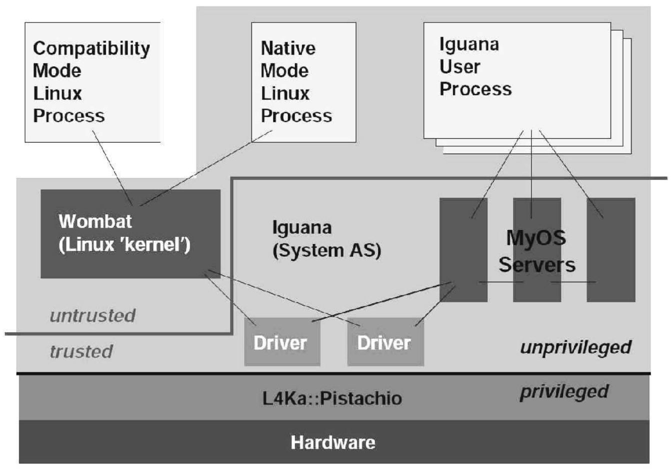


Figure 2.8: Iguana and Wombat

In the system consisting of Iguana and Wombat the Linux applications run in two different ways:

- native mode,
- compatibility mode,

as it is shown in Figure 2.8.

Native mode is a mode when Linux processes run inside the shared address space. It gives the applications all the advantages of running in the same address space, e.g. fast context switches, a full access to Iguana services, and it allows them to communicate with other Iguana applications including the Linux server. Applications running in this mode can use combined API of Linux and Iguana and can even offer services to the rest of the Iguana system. Disadvantages of this mode can be in some cases: limited address space size, unusual address space layout, and no support for `fork()`. This means that some Linux applications will not run in native mode without a significant porting effort [6].

Purpose of compatibility mode is to provide full binary Linux compatibility for applications running in this mode without the need to port them to the native mode. With the advantage of this comes one disability for such applications. They can only communicate directly with Wombat using Linux system calls and for communication with other separate spaces, Linux standart mechanisms, e.g. files, pipes, sockets have to be used. This causes context switch overhead between different address spaces. Applications are less integrated into the rest of the system.

Iguana

Due to the minimality of L4 microkernel, there is a need to have an environment build around it that would provide at least the basic system services. Iguana is such a base, designed for embedded devices that requires different approaches to a number of issues than in L4Linux. It provides services such as allocating and sharing memory, a memory protection model and its enforcement, general resource management, device driver framework. Next it supports an address space layout that leads to a dramatic reduction of context switching overheads on processors with virtually addressed caches (e.g. ARM7 and ARM9). Iguana seems to complement rather than hide underlying L4 API. The most important role of Iguana is that it provides underlying OS for Wombat.

This is a short description of Iguana packages which are used to provide services for Wombat as described in [26]:

l4kernel A version of L4 microkernel, based on L4Ka::Pistachio, that is designed for embedded systems.

ig_server Iguana server.

ig_init Iguana initialization program. It initializes the system starting any servers and user shell.

ig_naming Iguana naming server. It provides a simple, shared, flat namespace.

ig_serial Iguana serial server.

ig_trace Iguana trace server.

ig_timer A timer server for Iguana.

2.6.2 Implementation

At the time of this work only the compatibility mode for Linux applications was available. It has been discovered that not all Linux system calls are implemented in the prerelease version of Wombat. This makes it unusable to run on top of native Linux distribution, like it is possible with L4Linux. On the other side, it must be noted that the Kenge building environment used for L4 projects with Iguana is much more straightforward than L4Env. It mainly consists of scrips and configuration files written in Python programming language.

To port Linux on top of L4 microkernel, a new processor architecture was introduced similarly as in L4Linux. This was achieved by adding new directories `include/asm-l4` and `arch/l4`. The new `l4` architecture was implemented from scratch to keep it as architecture neutral as possible.

Architecture specific code resides in an appropriate architecture specific subdirectory, e.g. `arch/l4/sys-{arm,i386,mips}`. Ports to other architectures, like PowerPC and Alpha, are still in development. The implementation started with Linux kernel 2.5 series in late 2003. For the testing purposes of this work the prerelease version of Wombat with kernel 2.6.10 has been used. Not all Linux system calls were implemented during this work, making it impossible to run some planned macrobenchmarks.

System Calls and Exceptions

Implementation of a Linux process is designed as a single L4 task with only one running thread. This means that it is able to use L4 system calls, such as for L4's IPC communication. Even in compatibility mode to interact with other system's processes. One could argue that this is not possible according to the description of this mode presented in the subsection describing system architecture. Sending IPC messages to other than Wombat processes is restricted using security mechanisms offered by L4. It should be noted that applications running in compatibility mode are built for native Linux and do not know about L4.

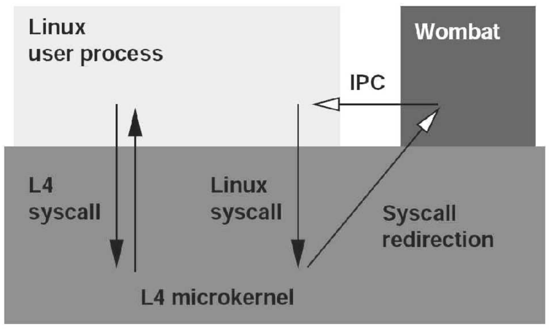


Figure 2.9: System call redirection

For handling native Linux system calls, Wombat uses a technique called system call redirection, or also trampoline. This is has to be done, because `l4` has different syscall numbers. The redirection mechanism works in three steps: (1) when a Linux user process performs a Linux system call, the L4 treats this as an exception, translates it and (2) forwards to Wombat as an IPC message. Wombat looks at this as at normal IPC message from user process, subsequently processes the Linux system call by invoking standart Linux kernel services, and (3) returns directly to the application process. On ARM processors different mechanism for system calls is used.

Scheduling

Similarly to L4Linux, two schedulers are involved: L4 kernel's internal scheduler, and normal Linux scheduler. For scheduling Iguana's and Wombat's threads is used L4's scheduler is used, while the normal Linux processes are scheduled by Linux scheduler. To meet the realtime scheduling criteria, realtime threads are normally given higher priority than Wombat, thus real-time part will always preempt Linux part of the system.

Only one single Linux user process is can run on one CPU. This is ensured by Wombat, in order to guarantee proper implementation of Linux scheduling. The purpose of special thread called timer thread running in Wombat with higher priority than all Linux processes is to maintain the Linux time slice. When it receives timeout corresponding to Linux timer tick, it wakes up and calls Linux scheduler to determine next process to run.

Device Drivers

There are three different types of device drivers that can be involved in the system running on Iguana and Wombat. Firstly, there are normal Iguana drivers running as user-mode servers inside their own protection domain. Second type are standart Linux drivers that can run unmodified on Wombat. When standart Linux drivers are used they can not be shared with other realtime applications running on top of Iguana because they reside in non-realtime part and thus do not meet all the necessary criteria. It should be noted that Linux drivers are not allowed to perform DMA, because this would break the security encapsulation of the Linux side of the system. The third type is shown in Figure 2.10 from [6]. Only one driver can control each device. The implementation of shared devices allows access for both, Iguana servers and Wombat. The device is owned by server implementing the proper driver. The other side, e.g Wombat, implements only a so called stub driver which forwards its requests to the proper driver. To ensure correct arbitration of resources, the proper driver is usually implemented as Iguana server.

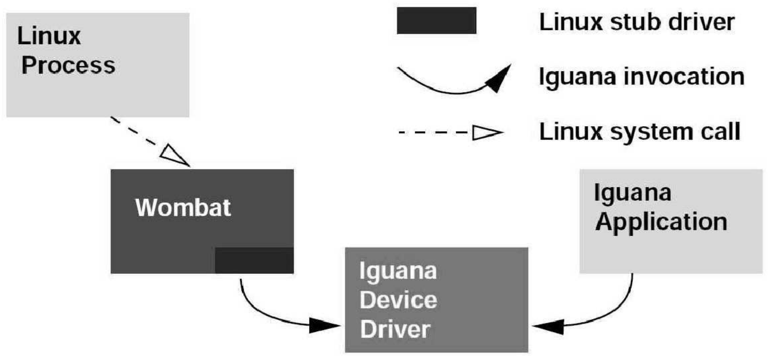


Figure 2.10: Shared device drivers

Chapter 3

Installation and Associated Problems

In this chapter we will discuss the installation of L4Linux and Wombat, which are Linux ports on top of L4 microkernel, and problems that we had to deal with together with decisions that we had to make. The goal of this phase was to get reasonably well working environment with installed L4Linux and Wombat on native hardware, more specifically on IA32 architecture. All of this had to be accomplished within the limited time span that was available for this assignment. It should be noted that this phase took much more time than it was expected. Absence of proper documentation, outdated information, hardware problems and many times the lack of knowledge made it really hard to make decisions when dealing with problems in very specific areas. This wouldn't be accomplishable without advises obtained from the mailing lists and people signed up. We suggest to everyone who wants to do research or development in this area to sign up for appropriate mailing lists as soon as possible and get help from the community.

Our installation process could be divided into few steps. Before the separate installation we had started to experiment with the available Live-CD from TU Dresden, called TUD:OS Demo CD, inside VMware virtualization machine to get an idea how the resulting system might look like. The virtualization tools (e.g. VMware, Xen, QEMU) help immensely while operating systems are developed, especially when there is a risk of damage on real hardware or of loss of data, not to mention numerous consuming system reboots.

Subsequently, we moved on to creating desktop Linux base system that we used for building and installation of final systems. The rest of this chapter describes the installation steps and issues we had to deal with.

3.1 Test-Bed Infrastructure

The environment, created for building and running the selected microkernel based systems, played an important role during the installation phase. Following subsections provide a short summary of hardware and software configuration, together with associated problems.

3.1.1 Hardware

Proper selection of hardware may reduce many problems during the development phase. Thus, we suggest selecting all hardware components carefully in order to avoid many troubles that may occur later on. During our research we did not have such a possibility, since we had only an old laptop at disposal at that time. The test system we had used for development consisted of a Pentium 4 CPU at 2.8GHz with 521KB cache size, 512MB of SDRAM, ATI Radeon 9000 IGP video card, and Realtek RTL-8139* ethernet. We decided to limit the system memory for L4Linux and Linux to 256MB. Thanks to the limitation we obtained more accurate results. The first hardware configuration was used solely for L4Linux.

As it became apparent later, the first configuration missed serial RS232 port, which is very helpful for kernel debugging with remote serial console. It was possible to use the final version of build system with Wombat and installed on real hardware only with the remote serial console. The way how to use the serial console is discussed further on. Another disadvantage of this system is the ethernet card. Following the first few successful installations of L4Linux we decided to set up a network, change Linux kernel configuration file appropriately and recompile kernel to support the above mentioned ethernet card. Hardware is detected successfully but it is not capable of any communication. Several experiments and rebuilds showed that there is already a problem most likely with our hardware configuration. Probably because the device is sharing IRQs with Universal Host Controller Interface (warrants further research). According to the experienced problems we would suggest to use an old NE2K PCI ethernet card for future research.

Without using a remote serial console we were not able to access and use system running Wombat. It should be noted that emulating Wombat in QEMU with simulated PCI VGA card was successfull. As far as we were aware, VGA should work on any IA32 machine, because A0000-BFFFF is always EGA/VGA video buffer (this warrants further research). To overcome this problem with insufficient hardware configuration, we decided to move our research to another hardware.

Second test system consisted of an AMD Athlon(tm) 64 X2 4200+ at 2.2GHz, 2GB RAM, PATA disk, Nvidia NX7300GT graphics card and desired serial port. With this setup we tried to use several different ethernet cards but we were not able to make them work. There was no time for deeper investigation. This test configuration was used only for performance testing of Wombat.

3.1.2 Software

We used three differnet Linux distributions during the installation phase: Fedora Core 6, Debian Etch, and Debian Sarge. Our first choice was Fedora Core 6. While this distribution has many innovatory features, it became apparent that it was not suitable for building L4/Fiasco and L4Linux. The best choice for starting with L4Linux is Slackware or Debian (Sarge or Etch). Using these two distributions should eliminate the additional installation of missing packages, e.g. in Fedora Core 6. So, for further building of L4Linux we've created environment based on Debian Etch distribution.

After the installation of the base distribution is successful we recommend to check the presence of the following packages: `openssh-client,openssh-server, gcc-3.4, cpp, g++-3.4, libncurses5-dev, libncurses5, doxygen, tk-brief, latex2html, nasm, gawk, hyperlatex, imagemagick, zlib1g-dev, bison, byacc, flex, automake1.9, autoconf2.13, fig2ps, transfig, fig2sty, fig2sxd, make, patch`.

Additional installation of missing packages avoids many future errors and possible confusion, whether there is a problem in downloaded L4 related source codes or in the building environment. During the first attempts we experienced both of them many times. In such situations we recommend to read through mailing lists and if the solution is not found ask other participants for help.

Another really important information is to pay attention which of the gcc and g++ versions are used for building. We highly recommend to use version 3.3.x or 3.4.x while building L4/Fiasco and L4Linux. Using gcc and g++ version 4.x.x leads to many compilation errors during the building process.

For building Wombat we need the Python version at least 2.3, SCons build system, the compiler must be gcc-3.3. Next we need to have QEMU for the simulation of the build images. There should be no problems with creation of such environment using Debian Sarge. The building of Wombat with Fedora Core 6 was successful after using the recommended toolchain. The toolchain was downloaded from: `https://www.ertos.nicta.com.au/downloads/i686-gcc-3.3.4-glibc-2.3.3-2006-06-02.tar.gz`.

3.1.3 Remote Serial Console

During the installation phase of Wombat we experienced problems with running it in VGA mode. Whether it was caused by some implementation issues in prerelease version of Wombat or by misconfiguration in our building environment was not disclosed by the deadline. We were running out of time planned for this phase, thus we decided to use remote serial console was made. The description of the serial console can be found at [13].

3.2 L4Linux

This section shortly summarizes the installation process of L4Linux on real hardware and associated problems we had to deal with through different stages of installation. It should be noted that it is not possible to include comprehensive information on all the topics that had to be solved during the installation process. The detailed description of the whole installation process is not the aim of this paper.This section should give the reader rough idea on where to find more detailed or up-to-date information about the installation process.

The installation can be divided into two steps. First L4 environment and Fiasco kernel should be installed and second L4Linux itself. One of the basic requirements is a good knowledge of Linux system, programming experiences, and successful installation of an environment as described. During the L4Linux installation process we had to use helpfull information from a *l4-hackers* mailing list, other used information sources are [10], [7], [29].

3.2.1 Creating Directory Structure

First, the directory structure, for instance in `/mnt/L4`, should be created. These modules have to be downloaded from SVN or CVS repositories: oskit, oskit10, l4, l4linux-2.6. Second, grub-0.97 source codes and patch grub-0.97-os.1.diff should be obtained. That would add to it `modaddr` command and some other features necessary for booting Fiasco kernel. The patch is available for download from personal web page of Adam Lackorzynski (`http://os.inf.tu-dresden.de/ãdam/grub/0.97/grub-0.97-os.1.diff.gz`). It should be noted that this patch changes symbol expansion. This causes problem with booting GNU/Hurd operating system with patched version of grub.

Desired directory structure should look as follows:

```
oskit/
oskit10/
l4/
l4linux-2.6/
grub-0.97/
```

Up-to-date instructions how to access repositories together with short module descriptions are available on the DROPS download page [30]. The source codes are understandably changing time to time and after solving one bug new regression might occur. The best place to start with solving building problems is to get the information on the mailing list.

3.2.2 L4 Environment and Fiasco

The purpose of the following section is to describe configuration of L4 environment Fiasco in order do get the working DROPS system for running L4Linux.

Before building Fiasco kernel and servers we need to configure the L4 environment and the L4 kernel itself. In the following text we propose that `/mnt/L4/` is our main directory where we have the desired directory structure, and `/mnt/L4/build` is our build directory through the whole building process. First we create the build directory in our main directory, e.g. executing command:

```
# make -C l4 config O=/home/L4/build
```

Configuration interface will appear, similar to the menuconfig of Linux kernel. Here we should change in the `Paths and Directories` section path to match our main directory. Next we checked in the `Compilers and Tools` section to use special C-compilers, specifically we used gcc and g++ version 3.4. Another option is to create symlinks for gcc and g++ to use 3.4 versions. In such case a no changes in compiler section were required.

Next step is the configuration of Fiasco kernel. Change directory to `l4/kernel/fiasco/` from our main directory and issue `'make menuconfig'` command. Fiasco kernel configuration menu will appear. If the symlinks were not used as described at the end of the previous paragraph, then in the `'Compiling and Building'` section the compilers had to be changed in the same way as before. Next you should check your processor type, e.g. in `/proc/cpuinfo`, and set it appropriately in section `'Target System Options'`. For the first try we left everything else as it was and changes were done after a successful build.

When we were experiencing problems this was the first place we looked at. To save the changes and exit input we typed `'x'`.

After doing the described configuration changes, we started with compilation of all packages. This was done by issuing this command from our main directory:

```
# make -C l4 O=/home/L4/build
```

At the end of the building process, we run into the problem with building documentation due to missing a `html.sty` file. When we were prompted to enter new filename we just typed capital `'X'`. At the end of a successful build process we had build Fiasco kernel together with other essential servers as described. The servers were in subdirectory `build/bin/x86_568/l4v2/`, the bootstrap in `build/bin/x86_586/` and the Fiasco kernel in `l4/kernel/fiasco/build/` directory.

3.2.3 Configuring and Building L4Linux

As it has already been mentioned, we used L4Linux version based on Linux kernel 2.6.17 for our final performance tests. The following section summarizes installation steps that had to be done in order to get running L4Linux on top of Fiasco kernel. This might differ with a newer version of L4Linux.

Configuration of L4Linux is similar to the configuration of Linux kernel. In the subdirectory `l4linux-2.6/` we issued `'make menuconfig'`. At the top of configuration menu was `L4Linux configuration` submenu. Here we had to change the `'L4 tree build directory'` to match our build directory. Next we went over the target architecture and in the `Stub drivers` section we checked only these options in order to use l4con or DOpE:

```
[*] Use the rtc server \newline
[*] Block driver for the generic_blk interface
[*] Framebuffer driver for l4con and DOpE (input/output)
[*] Support for the X Window System driver
[*] Pseudo serial driver for console
[*] Serial console support
```

Everyting else was left unchecked, and we were done with configuration changes in `L4Linux configuration` submenu. Next we had to configure the drivers. This was probably the most trickiest part of the configuration and took a lot of time. Finally we got one configuration at the *l4-hackers* mailing list and used it as the base for the creation of our own configuration file. We spent long time configuring and exploring config options before we were able to successfully compile the L4Linux kernel to match our hardware. Generally it should be safe to use nearly anything like drivers but we had to make sure not to enable features like ACPI, SMP, preemption, apic/ioapic, HPET, highmem, MTRR, MCE, power management [29] etc. After saving the configuration changes, the L4Linux was built with `make`. A successful build produced vmlinuz26 in the `l4linux-2.6` directory. Between the rebuilds `make clean` had to be used as the L4 part did not seem to be fully capable of recognizing configuration changes and could cause problems.

3.2.4 Grub and Final Setup

At this point we had all parts and we were ready to install them on native hardware. Before doing that we had to reinstall the actual grub with our patched version of grub. This was simply done as with any other program. From our main directory `cd` we did e.g.:

```
# cd grub-0.97
# patch -p0 < ../grub-0.97-os.1.diff
# ./configure ; make ; make install
# grub-install -v
```

We had to see something like 'GNU GRUB 0.97-os.1', otherwise something went wrong. After a successful installation of the patched grub we just installed the grub on the desired partition e.g. `grub-install /dev/hda`.

At that point we had to collect all the components and create final configuration files. The best way was to create a new directory in /boot/ and copy all necessary binaries into it. This was done e.g. with this sequence of commands :

```
# mkdir /boot/L4Linux
# cd /mnt/L4/build/bin/x86_586/l4v2/
# cp {bmodfs,con,dm_phys,events,l4exec,l4io,loader,log,names,roottask, \
rtc,sigma0,simple_ts,libld-l4.s.so,tftp,libloader.s.so} /boot/L4Linux/
# cd .. ; cp bootstrap /boot/L4Linux/
# cp /mnt/L4/l4/kernel/fiasco/build/main /boot/L4Linux/
# cp /mnt/L4/l4linux-2.6/vmlinuz26 /boot/L4Linux/
```

It should be noted that in the above example code `main` is actually the Fiasco kernel. The last step was to create a linux configuration file and add new boot entry into grub `menu.lst` file. Linux configuration file was created e.g. like this:

```
# cat > /boot/L4Linux/linux26.cfg << EOF
verbose 0
task ``vmlinuz26'' ``earlyprintk=yes mem=256M video=l4fb root=/dev/hda1''
all_sects_writable allow_cli
EOF
```

And the new entry in grub `menu.lst` configuration file was added like this:

```
# cat >> /boot/grub/menu.lst << EOF
title           L4Linux26/Fiasco con
# /boot on hda1
root (hd0,0)
kernel          /boot/L4Linux/bootstrap
modaddr         0x06000000
module          /boot/L4Linux/main -nowait -nokdb -serial_esc -comspeed \
115200 -comport 1
module          /boot/L4Linux/sigma0
module          /boot/L4Linux/roottask task modname ``bmodfs'' attached \
4 modules
```

```
module          /boot/L4Linux/events
module          /boot/L4Linux/names --events
module          /boot/L4Linux/log --events
module          /boot/L4Linux/dm_phys --isa=0x00800000 -v --events
module          /boot/L4Linux/simple_ts -t 300 --events
module          /boot/L4Linux/rtc --events
module          /boot/L4Linux/ore --events
module          /boot/L4Linux/l4io --noirq --events
module          /boot/L4Linux/l4exec --events
module          /boot/L4Linux/con --l4io
module          /boot/L4Linux/loader --fprov=BMODFS linux26.cfg
module          /boot/L4Linux/bmodfs
        module          /boot/L4Linux/vmlinuz26
        module          /boot/L4Linux/linux26.cfg
        module          /boot/L4Linux/libloader.s.so
        module          /boot/L4Linux/libld-l4.s.so
vbeset 0x117 506070
```

Now when we rebooted and selected `L4Linux26/Fiasco con` from grub menu, we saw shortly loading modules followed by Fiasco kernel and DROPS console. When the DROPS console started the L4Linux booted into the console window and if everything went fine were able to login as into native Linux.

3.3 Wombat

Installation of Wombat is much more straightforward than the installation of L4Linux. Before we found out the right way to achieve this goal, we had to solve many problems. In this section we assume that the required software version and tools as described in the section about software installation has been successfully installed. Without these requirements it is easy to get into problems when building the final image. In the rest of this section we are describing what we have done before actual installation of Wombat, and then we describe installation of Wombat on native IA32 hardware.

Before starting with actual installation of Wombat, we have tried to install Hello World project. This is a simple server running on top of Iguana based system and helps to understand the Kenge build system, which is used to build projects like Iguana. Kenge is based on SCons build system and thus uses mostly Python programming language. After successfull simulation of Hello World project in QEMU we have moved to installation of Wombat.

Wombat is not intended to be used standalone but as a part of Iguana project. Our second step was building Wombat as a part of Iguana. The iguana package is available for download from `http://www.ertos.nicta.com.au/software/kenge/iguana-project/devel/iguana-project--devel--1.1--version-0.tar.gz` has many bugs that makes it hard to install for newbie, few patches have to be applied in order to get a working image. We decided to try unsupported prerelease tarball, though we had a working image. As we were informed, it had to fix many bugs and do VGA, framebuffer, NE2K ethernet.

In the next few paragraphs we give instructions on installation of prerelease version of Wombat on real hardware. Note that at the end the Wombat will be able to boot on hardware but nothing at all will work with root file system that requires NPTL. We did this with Debian Sarge which does not require NPTL. Not all Linux system calls are supported in the prerelease tarball, as we found out later by looking into syscall header file `include/asm-l4/syscalls.h`.

First the prerelease tarball needs to be downloaded e.g.:

```
# wget --no-check-certificate \
https://www.ertos.nicta.com.au/downloads/prerelease.tar.bz2
```

and extract the archive to our main directory, from where all the changes will be done. Now we need to do some changes in configuration and fix some minor issues before the start to build the image.

There is `.conf` file which is the configuration file for SCons build system. It allows individual projects to specify command line arguments that change the behavior of the project. Configuration options in `.conf` can be overridden on the command line. For building our testing image we used SCons configuration with this content:

```
machine=‘‘pc99_vga’’
wombat=True
build_linux=True
toolprefix=‘‘i686-unknown-linux-gnu-’’
ltp=‘‘all’’
linux_apps=‘‘ltp’’
lmb=‘‘all’’
linux_apps=‘‘lmbench’’
```

This will build an image for IA32 architecture with support for VGA, it will include Wombat, busybox, basic set of lmbench tools. Next the size of build images has to be increased. We've found bug in `tools/bootimg.py` file, where the values assigned to `size` variable at lines 173 and 200 have to be doubled. Now we have to export path to the recommended toolchain and start building with SCons, e.g.:

```
# export PATH=$PATH:/opt/nicta/gcc-3.3.4-glibc-2.3.3/ \
i686-unknown-linux-gnu/bin/
# tools/scons.py
```

After successful build, there is an image for booting from USB removable device in `build/usb.img`. Rather than rebooting real hardware, we recommend to run the SCons again with parameter `simulate`, e.g.:

```
# tools/scons.py simulate
```

This will create a hard disk image file in `build/c.img` and will try to run it with QEMU. With some versions of QEMU this does not work and has to be done manually, e.g.:

```
# qemu -nographic -hda build/c.img
```

If everything went fine, the command line should appear after a short booting process, and we can install the image on real hardware. It is quite possible that some of step above will not work. In this case the best way to solve the problem is to look into the source code or find some help on the *kenge-users* mailing list.

Installation on native hardware is not too complicated. First the boot image has to be mounted e.g.:

```
# mount -o loop -o offset=32256 build/c.img /mnt/wombat
```

Next the content of the mounted image should be copied to the native boot partition. New entry has to be added into the native grub's `menu.lst` file simply by copying the entry from `menu.lst` on the mounted boot image. Now the machine can be rebooted into Wombat.

For some unknown reason we were not able to access the system directly. We had to access it with a remote serial console and `minicom` from some other machine. Since we had only one machine with serial, USB to serial (RS-232) converter had to be used. We had some problems with this because the first converter we used was able to communicate only at speed 19200Bps which forced us do some changes in source codes. Description of these changes would be too detailed and it is not appropriate for this type of document.

Described installation of Wombat on native hardware uses ramdisk and not real file system. Mounting this ramdisk allows to pass additional data files to Wombat and use them e.g. for testing. Another problem found is the fact that by default the system memory used by Wombat is only 45MB and that was not enough to perform our application tests and other macrobenchmarks. We found a way how to force Wombat to request more memory from the server which is responsible for memory allocation. Again for some unknown reason the server was not able to allocate more requested memory. We were not able to resolve this problem within the short time specified for this work.

Finally we tried to change parameters passed to `vmlinux` and force it to use real file system instead of ramdisk. This can be done in `iguana/init/src/init.c` at line 393 e.g. changing it to:

```
vmlinux root=/dev/hda1 console=tty0 console=ttyS0,115200n8
```

Now we have to rebuild it and copy new binaries from the mounted image to the Debian machine boot partition. Then all that has to be done is move `/lib/tls` out of the way. Many recent distributions are NPTL (Native POSIX Thread Library) based. The easiest way to check the system for this is with command:

```
# getconf GNU_LIBPTHREAD_VERSION
```

After reboot it should boot, but as we have already mentioned not all system calls are currently supported and thus most programs from Debian machine will not run.

Chapter 4

Methodology of Experiments

Microkernels are haunted by performance issues since the first performance results and their comparison with monolithic kernels appeared. In this chapter we describe our research methodology and selected benchmark tools used to measure a list of performances in order to give us a global view for performance comparison of L4Linux, Wombat and native Linux.

Most of the performance tests related to L4 microkernel were done in the past with native Linux and L4Linux kernel version 2.4. For comparison of L4Linux we used L4Linux-2.6.17 on top of Fiasco microkernel and on the monolithic side was Linux-2.6.18. The changes between Linux kernel version 2.6.17 and 2.6.18 are so small that according to our testing criteria they will not affect our experiments. The version of Wombat server was based on Linux kernel version 2.6.10 and compared to native Linux based on kernel 2.6.8.

To obtain comparable and reproducible performance results, the same hardware had to be used throughout all experiments. Due to hardware difficulties, we had to use two different hardware configurations for testing L4Linux and Wombat. This led us to two native Linux installations. One used for comparison with Wombat, and the other for L4Linux.

Before we started to gather actual results, we had to propose set of benchmarks and their purpose. Different benchmarks and Linux applications were used in order to get an impression on the overall performance. This has been done after the research on microkernel performance as is it described in the introduction. The Wombat is relatively new and we couldn't find any performance results except few comparisons provided by developers and mostly only for ARM architecture. The only performance tests for Wombat on IA32 architecture were for measuring context switch overheads and do not give us a complete picture of its performance [6].

Running each benchmark only once wouldn't produce reliable and conclusive results. To eliminate a measurement errors, every group of experiments has to be run many times to gather enough samples to allow us calculate average or select most reliable results.

The performance tests could be divided into two groups, microbenchmarks and macrobenchmarks. Microbenchmarks are used to analyze detailed behavior of all parts of the tested systems, while macrobenchmarks should give us perspective on the system's overall performance with synthetic and real applications. They are described more in detail in the next few sections of this chapter.

4.1 Microbenchmarks

In order to get a global and fair overview for performance comparison we decided to use the two most often used microbenchmark tools, *lmbench* and *AIM9*. This should allow us to compare our results with other results found on the Internet.

4.1.1 Lmbench

Lmbench is a collection of microbenchmark tools designed to test the performance of basic building blocks. These testing tools were designed to allow easy comparison of different system implementations. They test the most commonly found performance bottlenecks in wide range of system applications [22].

From the whole group of tools provided by this collection of benchmarks we've decided to choose only those tests that can be run on both Wombat and L4Linux. While there was no problem running any Linux console application on L4Linux, in Wombat we had only a small set of applications. Results from tests on both Linux ports can be compared to native Linux and vice versa. Specifically we will test process latencies related to:

- context switching,
- process operations,
- process communication.

Measuring latency of *context switching* is useful to observe time needed to save the state of one process and restore the state of another process. The program provided by lmbench to test latency of context switching is called `lat_ctx`. It allows us to change a number of *active processes* which are connected with Unix pipes and the pass token from process to process in a ring. Its second parameter is a *size of processes* which adds a artificial variable size 'cache footprint' to the switching process. The cost of the context switch then includes the cost of restoring user-level state. If the total size of all of the benchmark processes is larger than the processor cache size, the cost of the context switch includes cache misses.

The process operation benchmarks are used to measure the basic process primitives, such as timing simple entry into the operating system, creating a new process, and running a different program. For this purpose were used `lat_proc`, `lat_syscall`, and `lat_pagefault` tests.

The last group of test measured interprocess communication latencies. The first tool used is `lat_pipe` and it measures interprocess latency through pipes. Two processes are passing a token back and forth through a Unix pipe while no other work is done in the processes. Another tool `lat_unix` implements the same token-passing mechanism, but instead of pipes uses the Unix sockets. Tool, `lat_sig`, measures the time it takes to install and catch signals.

For more details on all mentioned tests please refer to manual pages [21] and [22]. They were used as an information source for the short description above.

4.1.2 AIM Independent Resource Benchmark XI

Second selected microbenchmark AIM Independent resource benchmark XI, short AIM 9. We chose AIM 9 because of its complex set of tests used to independently exercise and time different components of a UNIX-like computer system. Results are represented by absolute processing rates in operations per second. The tests could be divided according to their target system component into different groups:

- Arithmetic
- InterProcess Communication
- Disk/Filesystem I/O
- Library/System
- Memory and Process Management
- Function Calls
- Algorithmic Tests

This set of tests helped us to get an opinion on the final overhead added by L4 layer to the specific Linux subsystem and to applications running on top of it according to their character. It should be noted that it was possible to run this microbenchmark only for L4Linux and native Linux. The development phase of Wombat did not allow us to run this set of tests. For more details on AIM9 please refer to documentation attached to it [3].

4.2 Macrobenchmarks

What is the penalty of using L4Linux instead of native Linux? Analysis and comparison of selected macrobenchmark results should give us the answer. The next subsections describes synthetic AIM multiuser benchmark suite VII, for short AIM7, and then actual programs are presented. They were timed to get better opinion on performance slowdown of L4Linux. It should be noted that it was not possible to execute macrobenchmark tests on Wombat.

4.2.1 AIM Multiuser Benchmark VII

The AIM7 macrobenchmark is designed to measure the performance of multiuser computers. Capabilities of this suite were used to simulate multiuser system environment. It includes 55 basic tests and uses benchmarking technique called *Load/Mix Modeling* to simulate desired application workload. Different tests can generate a specific types of system load. These tests can exercise many basic functions of the multiuser system, such as system resources, library routines, the optimizing compilers, I/O subsystems, all levels of memory and so on. Configuration file specifies the mix, quantum and tests to be used during the testing. The whole process could be compared to cooking. The configuration file is the recipe and tests are the ingredients. During the testing the number of running tasks increases and thus the operation load increases. For each simulated operation load, the benchmark forks off one process. The type of process is defined by the configuration file.

The benchmark ends when the number of operation loads meets or exceeds the number of jobs that can be processed per minute.

The benchmark results include the following information:

- Jobs/Min - Number of jobs completed per minute for each simulated operation load.
- JTI - Job Timing Index which decreases with increasing operation load, and specifies how predictable the system is during the test.
- Real - Elapsed wall clock time.
- CPU - Used CPU time during operation load.
- Jobs/sec/task - The number of jobs processed per second.

These information were based on AIM7 documentation and for more information please refer to this document [4].

4.2.2 Application Tests

Another set of our macrobenchmark tests was examining resource usage of selected applications with the `time` utility. We recorded the elapsed real time between invocation and termination, the user CPU time, and the system CPU time. This should give us a better view of the overall performance from the user's perspective.

Firstly we measured the compilation time of L4 environment as it is described in the chapter about installation. Another utility was `ffmpeg` video converter which is a part of the Mediabench II specifications. With this tool we have converted mp3 formated files of different size into wav format. Text processing tool `sort` was used to sort a text file containing random dictionary entries. With `gzip` text files of differnet size were compressed.

4.2.3 Other Tests

As an additional performance test we planned to implement some user servers purely on top of L4 microkernel which would test the behavior of L4 microkernel. Finally we were able to successfully implement only a simple user space server doing arithmetic operations. We have discovered that implementation of libc library provided by Iguana is not fully implemented due to microkernel restrictions. There was no time left to study other Iguana libraries.

Chapter 5

Results and Evaluation

We promised to execute the same set of tests on L4Linux and Wombat and compare their performance with each other and then with native Linux. Due to the lack of information and due to the development phase of Wombat this was not possible. One of the reasons was that the current IA32 implementation of Wombat is still missing implementation of some Linux system calls. Consequently programs that require missing system call do not run under Wombat. The final version of Wombat was built with Busybox, which is combining many standard Unix utilities into a single small executable. This promised us to be able to use standart Unix text processing utilities. The system memory used by Wombat by default is set to 45MB which isn't enough to run reasonably large files. Execution time for small files would be too short to be accurate. Unfortunately the size of system memory in Wombat was not possible to change due to actual implementation of memory management server. This is planned to change in the near future.

Tests were executed separately for L4Linux, Wombat, and two Linux distributions . Results for L4Linux are compared to Debian distribution on the first hardware configuration based on Pentium 4. Performance of Wombat is compared to the Debian distribution on second hardware configuration based on AMD X2 processor. To get reliable and conclusive results the tests were run more times and the presented results are average values.

5.1 Microbenchmarks

5.1.1 Lmbench

First we measured latency of process operations and communication. Table 5.1 and 5.2 shows results for L4Linux and Wombat. Results are in microseconds. The Ratio column shows average values for L4Linux or Wombat divided by the value for native Linux and shows us the slowdown factor. The Delta is difference between native Linux and L4Linux or Wombat. Next we took results for L4Linux and Wombat and normalized them to native Linux. These values are shown as percentage in the overhead column. Accurate slowdown factors of L4 microkernel based Linux ports are shown in tables.

Comparing values for L4Linux and Wombat in the ratio column in table 5.1 and 5.2 shows us that the average slowdown factor of simple entry into the operating system is for Wombat smaller than for L4Linux. The Wombat system has faster read and write while

Test	Wombat	Linux	Ratio	Delta	% overhead
Process operations [microseconds]					
Simple syscall	1.7984	0.1390	12.9381	1.6594	1193.81
Simple read	2.2934	0.2934	7.8175	2.0000	681.75
Simple write	1.9874	0.2592	7.6663	1.7281	666.63
Simple stat	10.6236	1.5543	6.8350	9.0693	583.5
Simple fstat	2.5524	0.3280	7.7809	2.2244	678.09
Simple open/close	12.8226	2.5222	5.0840	10.3005	408.4
Procedure call	0.0086	0.0086	1.0078	0.0001	0.78
Process fork+exit	382.4361	77.8489	4.9125	304.5873	391.25
Process fork+execve	403.2258	84.6479	4.7636	318.5779	376.36
Process fork+/bin/sh -c	2743.1421	1966.1111	1.3952	777.0310	39.52
Pagefaults on /tmp/mb	2.9481	0.6494	4.5399	2.2987	353.99
Process communication [microseconds]					
Pipe	10.6482	3.0510	3.4900	7.5971	249
AF_UNIX sock	12.0589	5.3582	2.2506	6.7008	125.06
Signal handler install	2.5968	0.3854	6.7385	2.2114	573.85
Signal handler overhad	5.0493	1.1116	4.5422	3.9377	354.22

Lower is better

Table 5.1: Process operations and communication - Wombat

L4Linux has faster open/close file operation. Process creation slowdowns are quite close for both L4 systems.

Slowdown of process communication through pipes and Unix sockets is for both L4 systems similar, while signal handling is slightly faster in the L4Linux system.

Next, we measured context switch latencies for various numbers of processes and size. The results for the Wombat system and native Linux on the same hardware are in table 5.3 and 5.4. The first column shows number of connected processes of size that is indicated by the first row. Figure 5.1 shows the context switch time for Wombat on the left and native Linux on the right. Process size of 0k is denoted by continuous line, 1k by dot and dash line, 4k by ultrafine dashed line, and 16k is denoted by fine dashed line. Context switch time for less than twenty processes is faster in Wombat than in native Linux. This was unexpected results though [6] presents similar results.

Context switch results for the L4Linux system show again that context switch time in the microkernel based system is again smaller than in the native Linux. The results are in table 5.5, 5.6. In figure 5.2 the process size of 0k is denoted by continuous line, 1k by ultrafine dashed line, and fine dashed line denotes 4k process size.

Test	L4Linux	Linux	Ratio	Delta	% overhead
Process operations [microseconds]					
Simple syscall	2.8484	0.1450	19.6487	2.7477	1895.42
Simple read	3.1667	0.2873	11.0223	2.8302	985.1
Simple write	3.0694	0.2628	11.6812	2.7811	1058.4
Simple stat	4.8517	1.5869	3.0573	3.3060	208.33
Simple fstat	3.2561	0.3547	9.1807	2.8723	809.87
Simple open/close	8.5410	2.4924	3.4268	6.0275	241.83
Procedure call	0.0064	0.0069	0.9275	-0.0005	-7.25
Process fork+exit	575.2074	121.8115	4.7221	452.2995	371.31
Process fork+execve	666.4451	126.1348	5.2836	527.0828	417.87
Process fork+/bin/sh -c	7737.3333	3785.8333	2.0438	3945.1667	104.21
Pagefaults on /tmp/mb	5.4076	1.4557	3.7148	4.1137	282.59
Process communication [microseconds]					
Pipe	19.6858	5.8088	3.3890	13.8411	238.28
AF_UNIX sock	21.0634	9.5811	2.1984	11.2133	117.03
Signal handler install	3.5238	0.6216	5.6687	2.9187	469.52
Signal handler overhad	6.0654	2.2094	2.7452	3.8978	176.41

Lower is better

Table 5.2: Process operations and communication - L4Linux

5.1.2 AIM Independent Resource Benchmark XI

Table 5.7 shows average slowdown for each test category in AIM 9. This was calculated from tables 5.9 and 5.10. The ratio column shows performance percentage normalized to native Linux. The number of completed operations per second in native Linux is 100%, thus ratio lower than 100% means slower system. The slowdown is then the differnece between ratio value and 100%. From table 5.7 we can see that the lowest slowdown was for tests from arithmetic, function call, and algorithmic test group. While disk operations, inter-process communication, memory and process management had highest slowdown and thus applications with intensive use of this type of operations will apparently perform slower on the L4Linux system. Whether this is true will show our macrobenchmark results.

5.2 Macrobenchmarks

5.2.1 AIM Multiuser Benchmark VII

Systematic evaluation of L4Linux system was done using the AIM 9 macrobenchmark. The results are in Table 5.11 and Table 5.12. It shows well the multiuser systems perform under different application load [4]. The fourth column show us that under light load, starting with 1 task, the real time per benchmark run in L4Linux system is 4 times longer than in native Linux. This factor decreases and at load around 20 tasks the real time is in L4Linux 2 times longer. The AIM successively increases the load until the maximum

Procs	0k	1k	4k	16k	512k
1	0.545	0.130	0.000	0.110	2.870
2	0.350	0.333	0.410	0.417	17.027
5	0.597	0.620	0.903	1.133	16.790
10	0.703	0.777	0.870	1.427	17.523
20	1.140	1.267	1.510	3.047	--
40	2.237	2.420	2.823	5.400	--
60	2.680	2.783	3.467	5.733	--

Lower is better

Table 5.3: Context switch latency - Wombat

Procs	0k	1k	4k	16k	512k
1	0.075	0.165	--	0.110	--
2	0.750	0.753	0.797	0.833	9.490
5	1.010	1.083	1.147	1.673	10.460
10	0.993	1.033	1.217	1.573	10.760
20	0.947	1.010	1.197	2.600	11.363
40	1.240	1.257	1.827	4.990	13.847
60	1.547	1.740	2.810	4.960	12.183

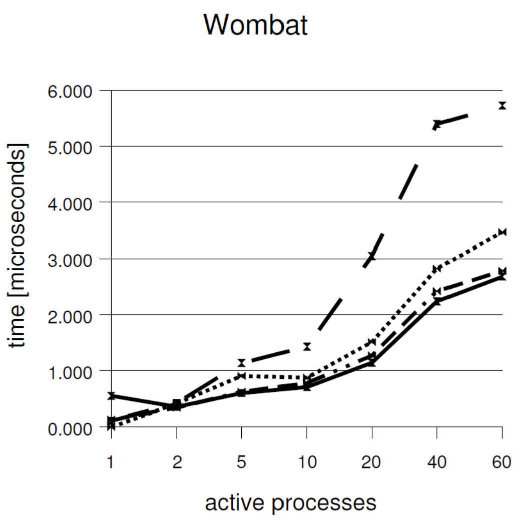
Lower is better

Table 5.4: Context switch latency - Native Linux (Wombat)

throughput of the system is determined. For this reason it stops at load 220 in L4Linux and native Linux continues increasing the load as it can be seen in the second part of Table 5.12.

Figures 5.4 and 5.3 show the number of completed jobs per minute depending on AIM load. The maximum load of jobs per minute was in L4Linux 634.6 while in native Linux it was 988.3 what is 1.5 times higher maximum load. Sustained load was almost 2 times higher in native Linux. The L4Linux system consumes 4 times more CPU time under lower load decreasing to approximately 2 times under the maximum load of L4Linux.

Similar AIM9 macrobenchmark were done in [14]. Presented results show that the typical penalties range from 5% to 10%. These tests were done more than 10 years ago with first version of Linux and L4Linux. Running these test on newer hardware and lates version of L4Linux and Linux kernel, show that the native Linux performs much better under heavy load.



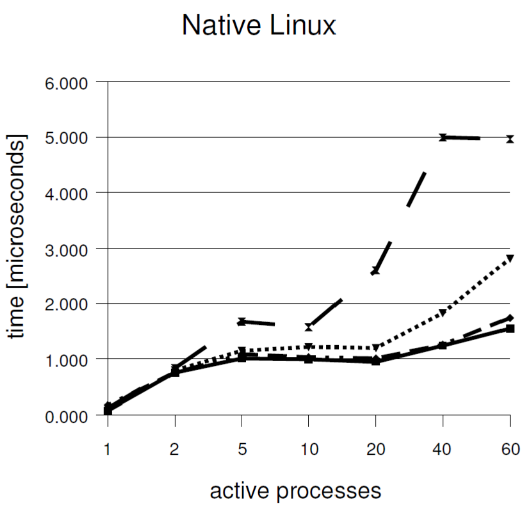


Figure 5.1: Graph of context switch overhead for process size 0k, 1k, 4k, 16k Wombat vs. native Linux

Procs	**0k**	**1k**	**4k**	**16k**	**512k**
1	0.060	0.140	0.090	9.590	18.910
2	0.910	1.043	0.940	38.063	100.850
5	1.270	1.367	1.470	33.347	95.820
10	1.277	1.537	2.453	32.420	93.895
20	1.847	2.683	6.077	41.460	94.660
40	3.807	5.303	9.977	43.113	101.420
60	5.403	7.103	11.313	42.757	103.530

Lower is better

Table 5.5: Context switch latency - L4Linux

5.2.2 Application Tests

The next set of tests were actual programs. The results are given in Table 5.8. This set of tests was again possible to execute only in L4Linux system and native Linux. Comparing the real time between invocation and termination the sort of text files on L4Linux was 2-8% slower than on native Linux for all file sizes. Compression of 20MB text file was 11% slower on L4Linux, while compression of 72MB file was as many as 73% slower. Conversion of mp3 audio files was approximately 25% slower on L4Linux system with smaller file sizes, while with 36MB file the time to convert mp3 file on L4Linux system was as many as 4.2 times longer. Simple calculation of Ludolf's number with Monte Carlo method was on L4Linux 10% slower what had been expected according to AIM 9 microbenchmark results.

Based on the microbenchmark results for L4Linux we suppose that the extremely high slowdown with larger files was due to slow disk and filesystem I/O operations together with slow memory management.

Procs	0k	1k	4k	16k	512k
1	0.020	0.015	0.020	0.020	--
2	1.827	1.960	2.053	44.933	125.115
5	2.327	2.560	2.680	49.210	135.230
10	2.240	2.407	3.310	40.363	116.640
20	2.527	3.087	5.847	50.277	116.420
40	3.670	5.133	9.630	45.560	105.320
60	4.970	6.640	10.693	55.467	139.075

Lower is better

Table 5.6: Context switch latency - Native Linux (L4Linux)

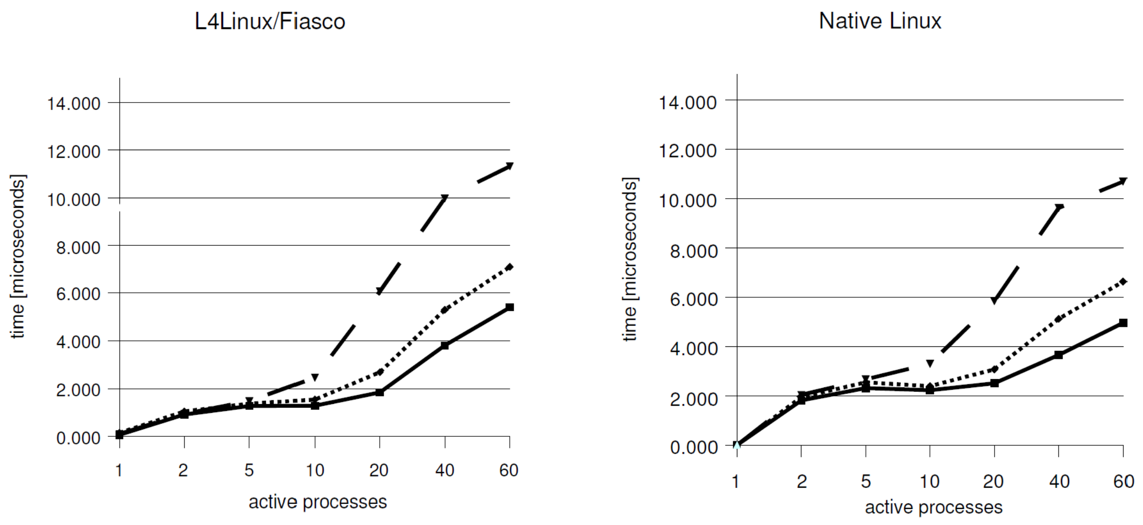


Figure 5.2: Graph of context switch overhead for process size 0k, 1k, 4k L4Linux vs. native Linux

Overall Average Slowdown	
Arithmetic	12.3%
Function calls	6.9%
Memory and Process Management	75.6%
InterProcess Communication	69.8%
Algorithmic Tests	11.0%
Library/System	30.3%
Disk/Filesystem I/O	65.3%

Table 5.7: L4Linux - overall average slowdown of each AIM 9 category normalized to native Linux

	Linux			**L4Linux**		
	Real [s]	*User [s]*	*Sys [s]*	*Real [s]*	*User [s]*	*Sys [s]*
Compilation	--	651.629	1763.094	--	1528.990	879.230
Sort 20M	22.196	21.381	0.212	23.790	23.610	0.180
Sort 10M	9.645	9.557	0.068	10.485	10.430	0.060
Sort 5M	4.567	4.128	0.036	4.673	4.580	0.060
Gzip 72M	7.466	7.261	0.140	12.932	12.671	0.178
Gzip 20M	2.685	2.623	0.048	2.998	2.900	0.100
Ffmpeg 36M	44.648	39.250	1.840	191.234	88.957	3.722
Ffmpeg 9M	12.624	9.609	0.420	15.572	14.330	0.970
Ffmpeg 4M	5.588	4.472	0.180	6.943	6.637	0.231
Monte Carlo	101.042	100.724	0.132	111.705	111.610	0.010
AIM 9	317.984	180.891	105.147	369.922	230.980	58.700

Lower is better

Table 5.8: Application tests

Test	L4Linux	Debian	Ratio	Delta	% slowdown
Arithmetic [Thousand Operations per second]					
add_double	186923	217425	86.0%	30502	14.0%
add_float	130693	218613	59.8%	87920	40.2%
add_long	2423762	2706586	89.6%	282824	10.4%
add_int	2437869	2701195	90.3%	263326	9.7%
add_short	1894488	2112000	89.7%	217512	10.3%
div_double	41749	46613	89.6%	4864	10.4%
div_float	41846	46706	89.6%	4860	10.4%
div_long	42772	47784	89.5%	5012	10.5%
div_int	43143	47784	90.3%	4641	9.7%
div_short	41593	46075	90.3%	4482	9.7%
mul_double	181176	201197	90.0%	20021	10.0%
mul_float	177865	196015	90.7%	18150	9.3%
mul_long	264906	293988	90.1%	29082	9.9%
mul_int	265417	293892	90.3%	28475	9.7%
mul_short	254075	281497	90.3%	27422	9.7%
Function Calls [Operations per second]					
fun_cal	168807920	168111776	100.4%	-696144	-0.4%
fun_cal1	170632871	186747410	91.4%	16114539	8.6%
fun_cal2	155950499	172608383	90.3%	16657884	9.7%
fun_cal15	70838356	78588423	90.1%	7750067	9.9%
Memory and Process Management [Operations per second]					
page_test	94820	382415	24.8%	287595	75.2%
brk_test	283896	2639920	10.8%	2356024	89.2%
exec_test	209	487	42.9%	278	57.1%
fork_test	1230	6427	19.1%	5197	80.9%
InterProcess Communication [Operations per second]					
shared_memory	70858	533652	13.3%	462794	86.7%
tcp_test	62425	168502	37.0%	106077	63.0%
udp_test	83585	312994	26.7%	229409	73.3%
fifo_test	106227	570938	18.6%	464711	81.4%
stream_pipe	103592	537924	19.3%	434332	80.7%
dgram_pipe	100958	491936	20.5%	390978	79.5%
pipe_cpy	110978	672814	16.5%	561836	83.5%

Higher is better

Table 5.9: Results of AIM 9 microbenchmark

Test	L4Linux	Debian	Ratio	Delta	% slowdown
	Disk/Filesystem I/O [Operations per second]				
link_test	47380	111149	42.6%	63769	57.4%
disk_rr	36887	138433	26.6%	101546	73.4%
disk_rw	32043	114003	28.1%	81960	71.9%
disk_rd	205413	886784	23.2%	681371	76.8%
disk_wrt	68334	177152	38.6%	108818	61.4%
disk_cp	59936	140469	42.7%	80533	57.3%
sync_disk_rw	318	697	45.6%	379	54.4%
sync_disk_wrt	89	805	11.1%	716	88.9%
sync_disk_cp	89	828	10.7%	739	89.3%
disk_src	37946	69711	54.4%	31765	45.6%
creat-clo	75848	247400	30.7%	171552	69.3%
dir_rtns_1	932270	1493013	62.4%	560743	37.6%
	Library/System [Operations per second]				
jmp_test	19923952	22202200	89.7%	2278248	10.3%
signal_test	69230	327944	21.1%	258714	78.9%
num_rtns_1	152842	157884	96.8%	5042	3.2%
trig_rtns	571992	632734	90.4%	60742	9.6%
string_rtns	3485	3892	89.5%	407	10.5%
mem_rtns_1	2488095	2784860	89.3%	296765	10.7%
mem_rtns_2	315564	350319	90.1%	34755	9.9%
sort_rtns_1	803	870	92.3%	67	7.7%
misc_rtns_1	4996	19754	25.3%	14758	74.7%
shell_rtns_1	50	115	43.5%	65	56.5%
shell_rtns_2	50	115	43.5%	65	56.5%
shell_rtns_3	50	115	43.5%	65	56.5%
ram_copy	803908422	882287101	91.1%	78378679	8.9%
	Algorithmic Tests [Operations per second]				
sieve	6	7	85.7%	1	14.3%
series_1	2238705	2486047	90.1%	247342	9.9%
matrix_rtns	939503	1039141	90.4%	99638	9.6%
array_rtns	360	401	89.8%	41	10.2%
new_raph	680319	755449	90.1%	75130	9.9%

Higher is better

Table 5.10: Results of AIM 9 microbenchmarks

Tasks	**Jobs/Min**	**JTI**	**Real**	**CPU**	**Jobs/sec/task**
1	35.5	100	163.8	12.7	0.59
3	89.4	96	195.3	39.6	0.50
5	170.7	99	170.5	47.6	0.57
7	245.9	99	165.7	60.4	0.59
9	343.1	98	152.7	53.1	0.64
11	384.0	98	166.7	72.7	0.58
13	391.4	96	193.3	90.9	0.50
15	446.6	95	195.5	85.6	0.50
19	464.7	94	238.0	127.7	0.41
23	502.4	95	266.5	148.6	0.36
31	522.9	95	345.0	209.1	0.28
39	551.0	93	411.9	252.0	0.24
56	575.3	94	566.6	337.2	0.17
63	570.9	96	642.2	365.5	0.15
77	597.0	95	750.7	428.0	0.13
107	628.7	94	990.5	604.9	0.10
119	622.1	93	1113.2	645.7	0.09
144	634.6	94	1320.7	768.2	0.07
198	620.5	94	1857.2	1057.7	0.05
220	609.2	94	2101.7	1212.1	0.05

Higher is better

Table 5.11: AIM multiuser benchmark VII results for L4Linux

Tasks	**Jobs/Min**	**JTI**	**Real**	**CPU**	**Jobs/sec/task**
1	143.4	100	40.2	3.6	2.39
3	217.5	92	79.5	8.3	1.21
5	428.8	97	67.2	12.9	1.43
7	634.3	96	63.6	18.0	1.51
9	705.7	94	73.5	22.9	1.31
11	834.5	94	75.9	28.1	1.26
13	850.8	93	88.0	33.1	1.09
15	966.1	91	89.4	37.6	1.07
19	943.2	89	116.0	47.1	0.83
23	932.0	92	142.1	56.5	0.68
27	988.3	93	157.4	66.0	0.61
35	894.2	92	225.5	85.7	0.43
38	932.0	90	234.8	93.4	0.41
44	960.9	92	263.8	108.2	0.36
56	907.0	90	355.6	137.4	0.27
61	948.2	91	370.6	149.2	0.26
71	918.9	90	445.1	172.0	0.22
81	947.1	91	492.6	194.6	0.19
102	912.2	90	644.1	244.3	0.15
111	919.2	91	695.5	263.9	0.14
129	868.0	93	856.1	305.5	0.11
147	863.3	91	980.8	348.1	0.10
185	864.2	91	1233.0	439.7	0.08
223	833.4	93	1541.3	529.6	0.06
261	808.9	93	1858.5	621.5	0.05
299	800.3	93	2152.0	712.6	0.04
381	739.7	94	2966.9	909.6	0.03
415	683.7	93	3496.1	991.9	0.03
489	625.3	94	4504.3	1172.3	0.02
563	564.2	94	5748.0	1357.2	0.02
564	562.6	95	5773.9	1358.2	0.02

Higher is better

Table 5.12: AIM multiuser benchmark VII results for native Linux

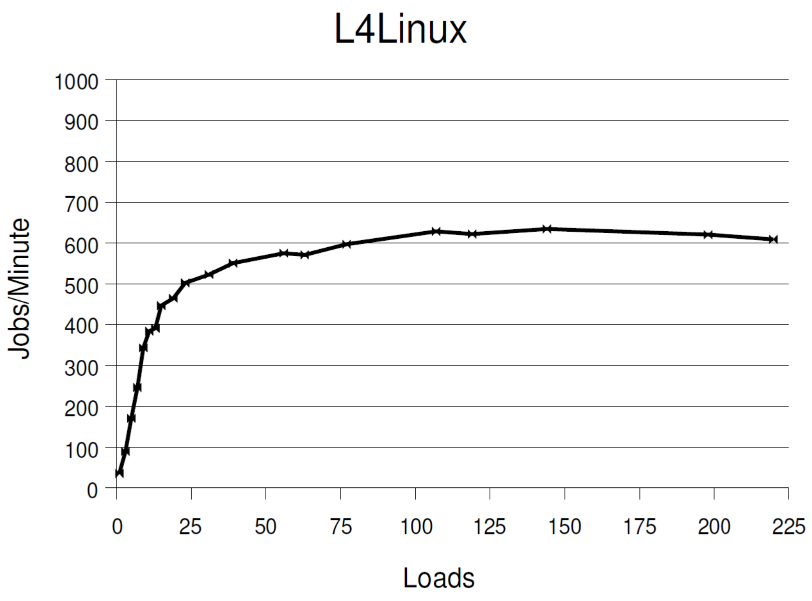


Figure 5.3: AIM multiuser benchmark VII results for L4Linux and native Linux

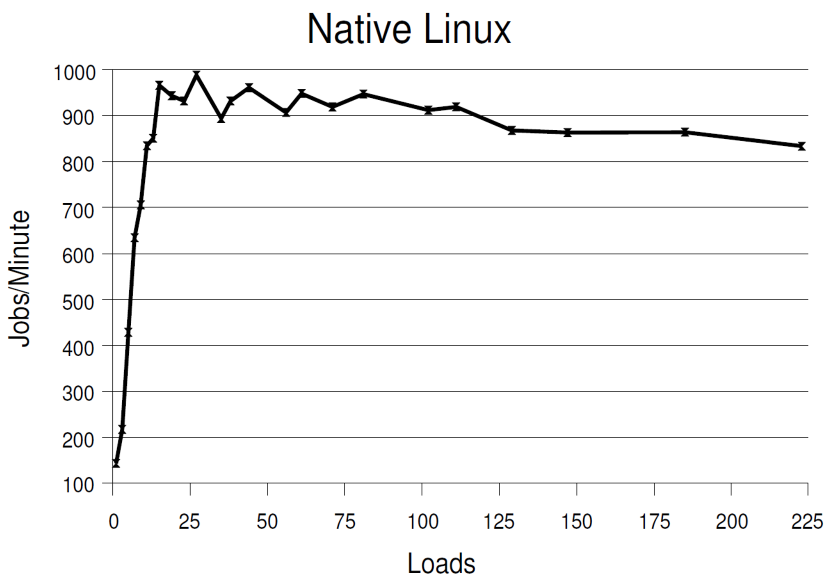


Figure 5.4: AIM multiuser benchmark VII results for native Linux

Chapter 6

Conclusion

The main contribution of this work is in familiarization with microkernel problem. We gained knowledge of theoretical concepts in the microkernel area. On the basis of the gained knowledge we decided to focus on two Linux ports on top of L4 microkernel, the L4Linux and Wombat. After many problems, we successfully installed selected operating systems based on two different implementations of L4 microkernel. The L4Linux uses Fiasco and Wombat runs on top of Pistachio-embedded implementation of L4 microkernel. Subsequently the performance of both systems was executed with proposed set of tests. Not all planned tests were feasible within Wombat due to incomplete implementation of Linux system calls.

The installation part of the overall work took much more time than we had expected. This would not be possible without appreciable help from the community around L4 microkernel. More specifically, we would like to thank Adam Lackorzynski, Hal Ashburner, Cheng Guanghui and many others. The practical part of this work was very time consuming. There was no time left to implement servers directly on top of L4 microkernel due to the lack of documentation and knowledge, and the fact that L4Linux and Wombat are still under development, and not to mention hardware requirements. This could bring new value to our work.

The performance results for L4Linux under heavy load and everyday usage for a few weeks drew us to the conclusion that it cannot compete with highly optimized monolithic kernels like Linux in the area of servers and personal computers. L4Linux seems to be a good base for many academic projects and research ideas, which can be even executed thanks to it. The performance of monolithic kernels like Linux is very high today. On the other side Wombat that targets on the area of embedded systems is much more likely to be successful. This proves the foundation of Open Kernel Labs and release of OKL4, commercial version of L4 microkernel and virtualization technology for embedded systems which is available together with Wombat [28].

For the further research in this area, we would suggest to build a small laboratory with appropriate hardware and accumulate a community of more experienced and knowledgeable researchers in all related areas which will focus on the embedded systems and OKL4 microkernel together with Wombat. We think that the use of L4 microkernel could be in embedded systems, secure and realtime applications, device drivers in user space, or distributed systems.

Bibliography

[1] Debian GNU/Hurd. URL: `http://www.debian.org/ports/hurd/`, 2007. (April 2007).

[2] Greg Gagne Abraham Silberschatz, Peter Baer Galvin. *Operating System Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[3] AIM Technology. *AIM Independent Resource Benchmark, Suite IX*, 1996.

[4] AIM Technology. *AIM Multiuser Benchmark, Suite VII*, 1996.

[5] Andrew S. Tanenbaum. Tanenbaum-Torvalds Debate: Part II. URL: `http://www.cs.vu.nl/ãst/reliable-os/` (April 2007), May 2006. (April 2007).

[6] Ben Leslie, Carl van Schaik, Gernot Heiser. *Wombat A Portable User-Mode Linux for Embedded Systems.* 2004.

[7] Cheng Guanghui, Nicholas Mc Guire. L4/Fiasco/L4Linux Kickstart. URL: `http://dslab.lzu.edu.cn/docs/publications/l4_kickstart.pdf` (April 2007), 2006.

[8] Cheng Guanghui, Zhou Qingguo, Nicholas Mc Guire, Wu Wenzhong. General Performance Assesement of the L4/Fiasco Micro-kernel. URL: `http://dslab.lzu.edu.cn/docs/publications/` (April 2007), 2006.

[9] Dipankar Sarma, Paul E. McKenny. Issues with Selected Scalability Features of the 2.6 Kernel. In *Proceeding of the Linux Symposium*, Ottawa, Ontario, Canada, July 2004.

[10] Frank Menhert, Jork Löser, Ronald Aigner. Building DROPS HOWTO. URL: `http://os.inf.tu-dresden.de/l4env/doc/html/drops-building/`. (April 2007).

[11] Gäel Le Mignot. The GNU Hurd. URL: `http://kilobug.free.fr/hurd/pres-en/abstract/html/`. (April 2007).

[12] Gernot Heiser. L4 User Manual. URL: `http://l4hq.org/docs/manuals/l4uman.pdf`, 1999. (April 2007).

[13] Glen Turner, Mark F. Komarinski. Remote Serial Console HOWTO. URL: `http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/`, March 2003. (April 2007).

[14] Jochen Liedtke Hermann Härtig, Michael Hohmuth, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, St. Malo, France, October 5–8 1997.

[15] Ihor Kuz. L4 User Manual NICTA L4-embedded API. URL: `http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/userman.pdf`, October 2005. (April 2007).

[16] Jochen Liedtke. Construction of Microkernels. URL: `http://i30www.ira.uka.de/teaching/coursedocuments/30/mkc-summary_felixhupfeld.html`, 2000. (May 2007).

[17] Andrew S. Tanenbaum Jorrit N. Herder, Herbert Bos. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, Jan 2006.

[18] L4hq. Kernel APIs. URL: `http://l4hq.org/kernels/` (April 2007).

[19] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2*, May 2003. URL: `http://l4ka.org/projects/pistachio/l4-x2-r5.pdf`. (April 2007).

[20] Adam Lackorzynski. L4Linux Porting Optimizations. Master's thesis, TU Dresden, 2004. URL: `http://os.inf.tu-dresden.de/papers_ps/adam-diplom.pdf`.

[21] Larry McVoy. Lmbench man pages. URL: `http://lmbench.sourceforge.net/man/index.html`, December 2006.

[22] Carl Staelin Larry W. McVoy. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

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

[24] Jochen Liedtke. Microkernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Seattle, WA, October 1996.

[25] Jochen Liedtke. Towards real microkernels. 39(9):70–77, September 1996.

[26] NICTA. Iguana Project. URL: `http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/`. (April 2007).

[27] NICTA. About Microkernels. URL: `http://www.ertos.nicta.com.au/research/l4/microkernels.pml` (April 2007), Jun 2006.

[28] Open Kernel Labs. OKL4 and Microkernel Technology. URL: `http://www.ok-labs.com`. (May 2007).

[29] Operating System Group Dresden. Building L4Linux-2.6 and requisities. URL: `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`, March 2007. (April 2007).

[30] Operating System Group Dresden. DROPS. URL: `http://os.inf.tu-dresden.de/drops/download.html`, March 2007. (April 2007).

[31] Sebastian Schönberg. A User Mode L4 Environment. 2001. 2nd Workshop on Microkernel-Based Systems.

[32] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park. Flexible Access Control Using IPC Redirection. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 29–30 1999.

[33] TU Dresden. Lecture on Microkernel Based Operating Systems. URL: `http://www.inf.tu-dresden.de/index.php?node_id=1314&ln=en` (April 2007), April 2007.

[34] TUDOS. DROPS - The Dresden Real-Time Operating System Project. URL: `http://os.inf.tu-dresden.de/drops/`, Jun 2006. (April 2007).