Monolithic kernel vs. Microkernel

Benjamin Roch
TU Wien
phibre@gmx.net

ABSTRACT

This document explains the two main kernel architectures of operating systems: the monolithic kernel and the microkernel. Starting with an introduction about the term "kernel" itself and its meaning for operating systems as a whole, it continues with a comparison of benefits and disadvantages of both architectures, rounded up by a list of popular implementations.

1. INTRODUCTION

The kernel is the indispensable and therefore most important part of an operating system. Roughly, an operating system itself consits of two parts: the kernel space (privileged mode) and the user space (unprivileged mode). Without that, protection between the processes would be impossible. There are two different concepts of kernels: monolithic kernel and μ-kernel (microkernel). The older approach is the monolithic kernel, of which Unix, MS-DOS and the early Mac OS are typical representants of. It runs every basic system service like process and memory managment, interrupt handling and I/O communication, file system, etc. in kernel space (Figure 1). It is constructed in a layered fashion, built up from the fundemental process managment up to the interfaces to the rest of the operating system (libraries and on top of them the applications). The inclusion of all basic services in kernel space has three big drawbacks: the kernel size, lack of extensibility and the bad maintainability. Bugfixing or the addition of new features means a recompilation of the whole kernel. This is time and resource consuming because the compilation of a new kernel can take several hours and alot of memory. Everytime someone adds a new feature or fixes a bug, it means recompilation of the whole kernel.

To overcome these limitations of extensibility and maintainability, the idea of μ-kernels appeared at the end of the 1980's. The concept (Figure 2) was to reduce the kernel to basic process communication and I/O control, and let the other system services reside in user space in form of normal processes (as so called servers). There is a server for managing memory issues, one server does process managment, another one manages drivers, and so on. Because the servers do not run in kernel space anymore, so called "context switches" are needed, to allow user processes to enter privileged mode (and to exit again). That way, the μ-kernel is not a block of system services anymore, but represents just several basic abstractions and primitives to control the communication between the processes and between a pro-

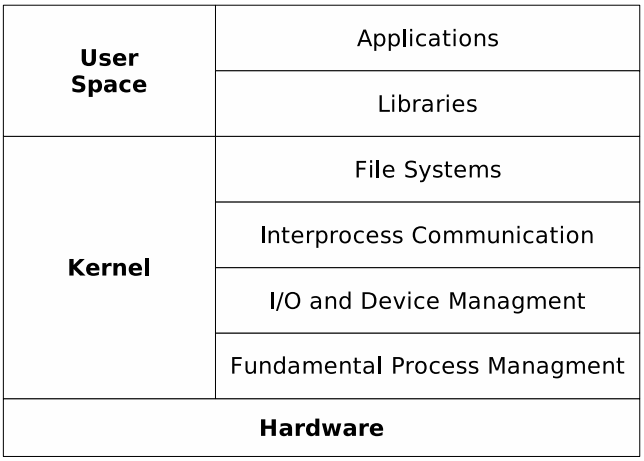

Figure 1: Monolithic kernel based operating system

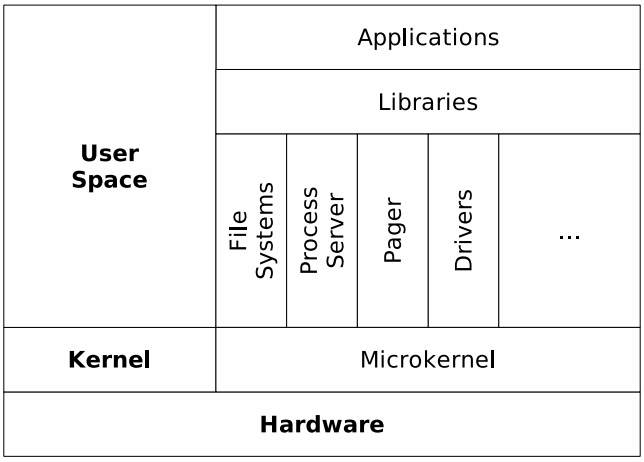

Figure 2: Microkernel based operating system

cess and the underlying hardware. Because communication is not done in a direct way anymore, a message system is introduced, which allows independent communication and favours extensibility (see 2.1 for more details).
Currently, there are two different generations of μ-kernels. The first generation was a more or less a stripped-down monolithic kernel. Because of performance drawbacks concerning process communication, several system services like device drivers, communication stacks, etc. found their way back into kernel space. This resulted in an even bigger kernel than before, which was slower than its monolithic counterpart.
Research in the field of μ-kernels proove, that it is not the best solution to create a hybrid kernel [1] , but a pure microkernel, which has to be very small in size. So small, that it fits into the processor's first level cache as a whole. Second generation μ-kernels like the L4 are highly optimized, not just referring to the processor family, but also to the processor itself [2] , which results in a very good I/O performance.

2. COMPARISON BETWEEN BASIC CONCEPTS OF THE TWO APPROACHES

This section introduces the basic concepts of operating systems, with their realizations in the two different architectures. For more information on operating systems, see e.g. [5].

2.1 Inter Process Communication

A process means the representation of a program in memory. It may consist of smaller, programmer defined parts called "threads". Threads allow virtually parallel execution of different sections of a program. A thread is the smallest unit[3] of executable code. A process can consist of several threads. In the next sections a thread and a process are meant to be equal, if not otherwise stated.
The earliest concept of inter-process communication (IPC) is called signals. It is widely used in Unix systems. Signals are predefined numerical constants, e.g. KILL, STOP, etc., which are sent to a process by a user, the operating system or another process. The signals are received by so called signal handlers, simple procedures, which belong to each process. This system is fast, but the problem is, that the signals have to be predefined numbers. Existing signals cannot be changed, because then, processes would react a different way than expected. New signals have to be standardized, which is too tedious for implementing just a single application.
Another solution for communication are the so called sockets. A process binds itself to one socket (or more), and "listens" to it, i.e. from then on, it can receive messages from other processes. Most of the sockets are full duplex [4]. The owner of a socket, i.e. the process which is bind to a socket, is also called server, while the other processes are called clients. Because messages sent through sockets are not limited to numerical values, the information could be more than just control signals. There is no limitation upon extensibility, as long as the server knows what the received messages mean.
A system more powerful than sockets are message queues. Built as a fifo queue, a message queue stores all incoming messages, sent by other processes and sorts them, based on their priority. One process can have more than one message queue, with every message queue being responsible for different kinds of messages.
While monolithic kernels use signals and sockets to ensure inter process communication, the μ-kernel approach uses message queues. It grants, that all parts of the system are exchangeable.
System components of the monolithic kernels are somewhat "hardwired". This prevents exstensibility. The first μ-kernels poorly implemented the IPC and were slow on context switches. [4] presents new concepts to overcome these performance lacks.

2.2 Memory Managment

Monolithic kernels implement everything needed for memory managment in kernel space. This includes allocation strategies, virtual memory managment, and page replacment algorithms (Figure 3).

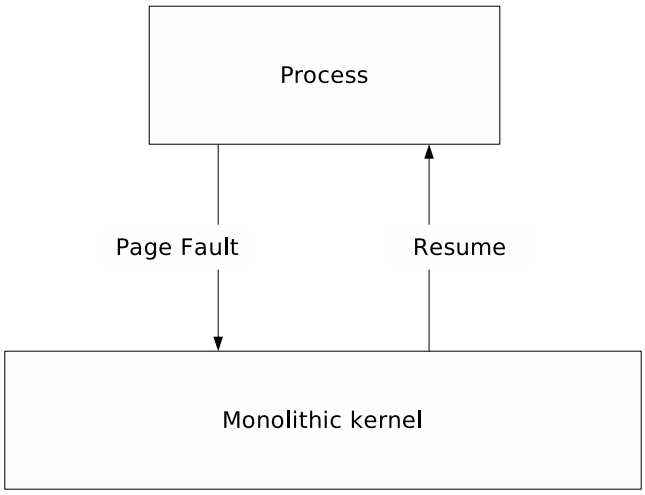

Figure 3: Memory managment of monolithic kernels

First generation μ-kernels delegate the memory managment to user space, controlling just the basic access rights (Figure 4). One of the servers is responsible for managing page faults and reserving new memory. Every time a page fault occurs [5], the request has to take the way through the kernel to the pager. The pager must enter privileged mode to get access to memory and get back to user mode. Then it sends the result back to the triggering process (again through the kernel). The whole procedure to handle page faults or reserve new memory pages is tedious and time consuming.

[1] A hybrid kernel is a mixture of monolithic and μ-kernel design.
[2] Intel has got a processor family which is called "Pentium". The optimization would not just differentiate between common intel x86 and Pentiums, but more fine grained, i.e. between a pentium II and a pentium III.
[3] With exception of a single instruction.
[4] Full duplex means communication in both directions: read and write at the same time.
[5] A page fault is generated, when the requested memory page is not in memory but swapped out on harddisk.

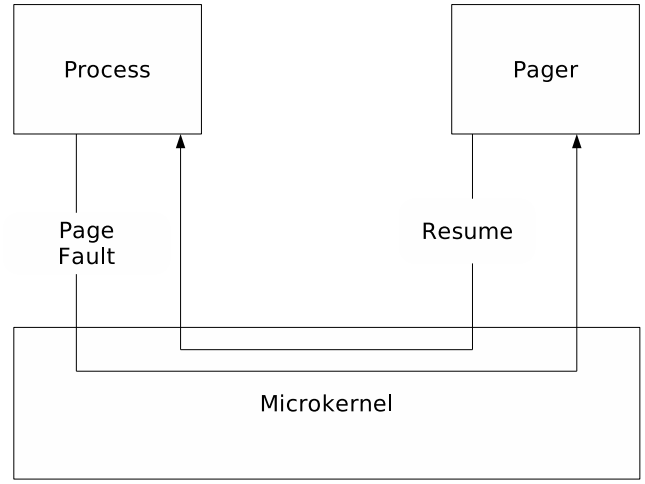

Figure 4: Memory managment of 1st-generation μ-kernels

To solve the performance loss, second generation μ-kernels have more refined strategies of memory managment, e.g. L4 (Figure 5). With L4 every process has got three memory

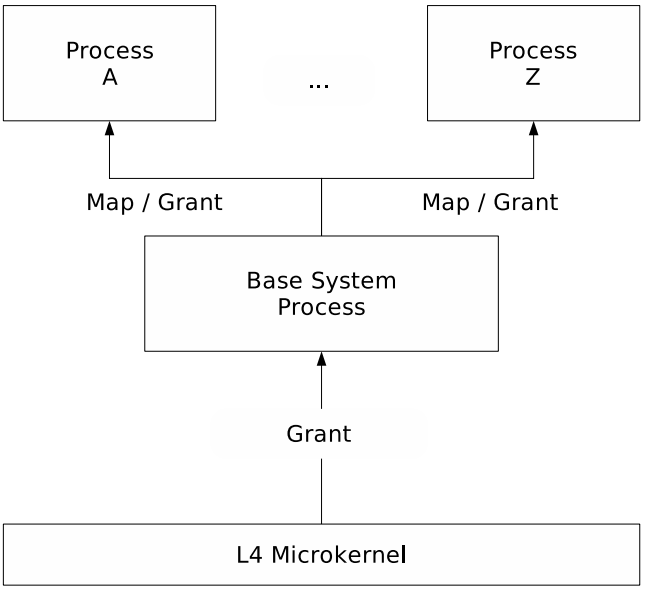

Figure 5: Memory managment of the L4

managment primitves: map, grant and flush. A process maps memory pages to another process if he wants to share these pages. When a process grants pages to another process, he cannot access them anymore, and they are under the other process' control, as long as the granting process does not flushes them. Flushing regains granted and mapped memory pages. This system now works as follows: The μ-kernel reserves the whole system memory at startup to one process, the base system process, which resides (like all other processes) in user space. If a process needs memory, he doesn't have to take the way through the kernel anymore, but directly asks the base system process. Because every process can only grant/map/flush the memory pages it owned before, memory protection still exists. That way the overhead of context switches is reduced to a minimum and performance is increased.

2.3 Security and Stability

The protection of system processes from being modified by the user or other processes is an important feature of the kernel. With the introduction of multitasking (and multithreading) new problems arised, concerning isolation of memory and processes. These problems include issues like race conditions, memory protection and system security itself. The kernel must be able to grant that in case a process cracks down, the system's performance will not be influenced. This is a more or less simple task if we talk about processes, which run in user space. But what happens, if a process crashes inside the kernel? Because of the "hardwiring" of system processes and the resulting dependency of the monolithic approach, it is assumable that other processes will also crash, resulting in a system-wide halt.
Excluding system processes from kernel space is a way to overcome these problems. Another argument for a true μ-kernelis its codesize. It is easier to ensure the correctness of a small kernel, than a big one. That way, stability issues are simpler to solve with that approach.

2.4 I/O Communication

I/O communication works through interrupts, issued by or sent to the hardware.
Monolithic kernels (and most of the first generation μ-kernels) run device drivers inside the kernel space. Hardware interrupts are directly handled by kernel processes. To add or change features provided by the hardware, all layers above the changed layer in the monolithic kernel also have to be changed in the worst case.
The concept of so called modules was introduced to achieve more independence and seperation from the kernel. One module represents (parts of) a driver and is (un)loadable during runtime. That way, drivers which are not needed by the system, are not loaded and memory is preserved. But kernel modules are still binary kernel-dependent. This means, modules working with monolithic kernel of generation A (e.g.: Linux 2.4.0) are not granted to cooperate with its successor (e.g.: Linux 2.4.2). Source compatibility is often assured, but not always. If concepts change too much inside the monolihic kernel, modules need not just a recompilation, but a complete code adaption.
The μ-kernelapproach doesn't handle I/O communication directly. It only ensures the communication. Requests from or to the hardware are redirected as messages by the μ-kernelto the servers in user space. If the hardware triggers an interrupt, the μ-kernelsends a message to the device driver server, and has nothing more to do about it. The device driver server takes the message and sends it to the right device driver. That way it is possible to add new drivers, exchange the driver manager without exchanging drivers or even exchange the whole driver managment system without changing any other part of the system. [3] shows that it is not desirable to put the drivers into kernel space, as it was done by the first generation of μ-kernels, to get acceptible performance. That way the size grows and the kernel cannot be fully held in the processor's cache memory.

2.5 Extensibility and Portability

Exstensibility is the most prominent fact for μ-kernels. It is, beside its size, one of the biggest difference to monolithic kernels.
Adding new features to a monolithic system means recompilation of the whole kernel, often including the whole driver infrastructure. If you have a new memeory managment routine and want to implement it into a monolithic architecture, modification of other parts of the system could be needed.
In case of a μ-kernel the services are isolated from each other through the message system. It is enough to reimplement the new memory manager. The processes which formerly used the other manager, do not notice the change. μ-kernels also show their flexibility in removing features [6] . That way a μ-kernel can be the base of a desktop operating system, as well as of realtime appliancies in single chip systems. On the other hand μ-kernels itself must be highly optimized for the processor they are intended to run on. It was shown, that it is not sufficient to just introduce a "compatibility layer" between kernel and processor as it was done with first generation μ-kernels. That way, μ-kernels are kept machine independent and could be easily be ported. Unfortunately, this approach prevented those μ-kernels from achieving the necessary performance and thus flexibility. [2] shows, that the introduction of such a layer between the kernel and processor has several implications:
i) Such a μ-kernel cannot take advantage of specific hardware.
ii) It cannot take precautions to circumvent or avoid performance problems of specific hardware.
iii) μ-kernels form the lowest layer of operating system. Therefore even the algorithms used inside the μ-kerneland its internal concepts are extremely processor depenedent.

3. IMPLEMENTATIONS

This is just a short overview presenting implementations of monolithic kernels, μ-kernels and hybrids. See the footnotes for links to more detailed information.

3.1 Monolithic kernel

3.1.1 GNU/Linux

GNU/Linux[7] is a free available, open source implementation of unix, developed by thousands of individuals. It is a typical representant of a monolithic kernel. Continously enhanced, it often changes its structure. Changing parts of the kernel means complete recompilation.
All system functions, including the whole process and memory managment, process and thread schelduling, I/O functionality and drivers are implemented in kernel space. The I/O communication, provided by so called modules, which can be inserted and removed during runtime, are built against the kernel, i.e.: If the kernel changes, the set of modules changes too. The estimated size of an average monolithic kernel is about twenty to thirty megabytes resulting in a tedious maintenance process.

3.2 Hybrid kernel

3.2.1 Mach

Mach[8] is a μ-kernel of the first generation, designed and developed at the Carnegie Mellon university. It represents the base, among others, of Next, Mac OS X, and built the foundation for alot of other μ-kernels designs. It was thought as a small-sized highly portable kernel which includes just a minimum set of kernel functions. The poor performance Mach showed in comparison to monolithic kernels led to the assumption, that μ-kernels cannot be fast. But second generation μ-kernels proved, that the lack of performance came due implementation issues, not due the μ-kerneldesign itself. Several mistakes of the first generation μ-kernels (e.g. Mach) can be pointed out:
i) To ease portability, the designers introduced an additional layer between the kernel and the CPU. That way, only the layer should be optimized for a given processor, but this approach turned out to be wrong [3].
ii) Because of poor performance, device drivers were put back into kernel space. This resulted in a bloated kernel, which couldn't reside in processor cache anymore.
iii) Mach's Inter-process communication is too tedious and time-consuming.

3.2.2 Windows NT

Microsoft introduced the kernel for their Windows NT[9] at the beginning of the 1990s. It was planned to be a μ-kernel, but due lack of performance, Microsoft decided to put alot of system services back into kernel space, including, among others, device drivers and communication stacks. This bloated the kernel and it became bigger than most monolithic kernels were that time.
NT (including the kernel) is designed as an object-oriented operating system. Therefore, all basic structures, like processes, threads, device drivers, and others are implemented as objects, handled by an object manager. The kernel talks to the hardware through a so called Hardware Abstraction Layer (HAL), which favors porting to other system architectures.

3.3 Microkernel

3.3.1 QNX

QNX[10] ("Quick Unix") is the most popular pure μ-kernel-based operating system for realtime applications. Realtime applications emphasize on predictability and stability. Examples of realtime appliancies are embedded systems like microwaves, dishwashers, car safety systems, cell phones, etc. Primary targeted to the embedded market, it is also available as desktop version.
Only the most fundamental primitves like signals, timers and scheduling reside inside kernel space resulting in a just 64 kilobyte-sized kernel. All other components, e.g.: protocol stacks, drivers and filesystems, run outside the kernel. All processes communicate via a single virtual messaging bus, that lets you plug in or plug out any component on the fly. The kernel of QNX (called neutrino) is posix complient, implemented in C and can be therefore easily tailored to different platforms and operating systems.
Developers can easily strip down the QNX kernel and remove unwanted functionality, e.g.: memory protection, which

[6] I.e.: Less security, little amount of drivers, ...
[7] http://www.gnu.org
[8] http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html
[9] http://www.microsoft.com
[10] http://www.qnx.com

is implemented as a module. If the target application(s) don't need the property, it can be easily removed, without editing any source code. QNX allows running, testing and debugging drivers during runtime, due their kernel space independency. If you want to know more about QNX, please refer to [1].

3.3.2 L4

The L4[11] μ-kernel was implemented at the TU Dresden by the Systems Architecture Group in cooperation with the IBM Watson research center. It belongs to the second-generation of μ-kernels. It prooved together with the QNX neutrino kernel, that μ-kernels cn be as fast as their monolithic counterparts allowing easy extensibility.
The performance is reached by the small size (12 kilobytes of code) and optimized inter-process communication (IPC). Due just three basic abstractions [12] , and seven system calls, on top of these abstractions, which are implemented in kernel space, allowing the L4 to completely residing in the processor's first-level cache. All other functions, like memory managment, device drivers, interrupt handling, protocol stacks, etc. reside in user space. That way, the L4 just controls access to the hardware and does basic thread managment. Hardware interrupts are forwarded to the user space as messages. The kernel is not involved in processing them. Memory managment is completely done in user space and fast address space switches are assured through processor-optimized code.

4. CONCLUSION

L4 and QNX, have proven that speed is not an argument against μ-kernels anymore. Their additional exstensibilty and portability predestines them for different applications reaching from embedded to desktop systems. They are more easily maintainable than their monolithic counterparts. Due the independency of the different parts made possible through message passing, μ-kernel systems can be easily extended and modified. Inter-process communication can be made faster by clever communication algorithms, often excluding the μ-kernel.
To achieve this perfomance, μ-kernels must be directly optimized for the processor, where they are intended to run on. That way, the full potential of a processor can be used directly.
μ-kernels allow to run multiple operating systems concurrently (Figure 6).

This implies new potential. Not every operating system needs to implement its own device drivers or communication stacks (Figure 7).
An additional idea would be to completely integrate the μ-kernel into the processor's architecture. This would grant even better performance and the best optimization for each processor (Figure 8).

5. REFERENCES

[1] Frank Kolnick. The qnx 4 real-time operating system. Jul 2000.

[11] http://os.inf.tu-dresden.de/L4/use.html
[12] The abstractions are: threads, address spaces, and IPC

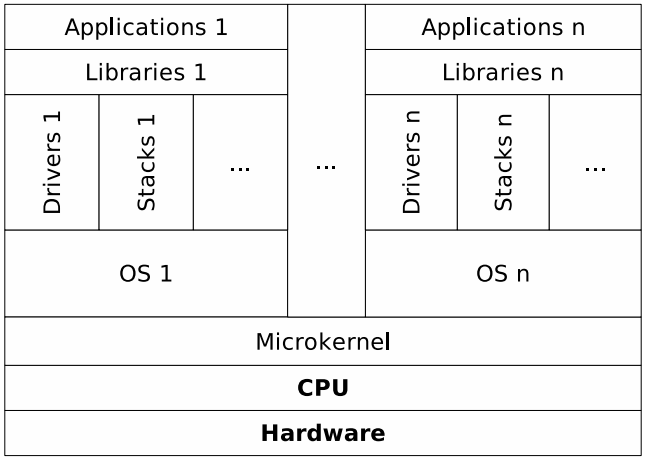

Figure 6: Multiple operating systems running simultaniousley on top of a μ-kernel

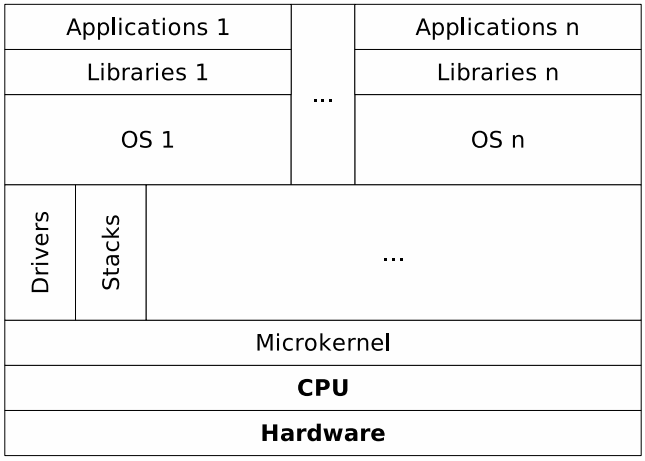

Figure 7: Multiple operating systems on top of the same base services

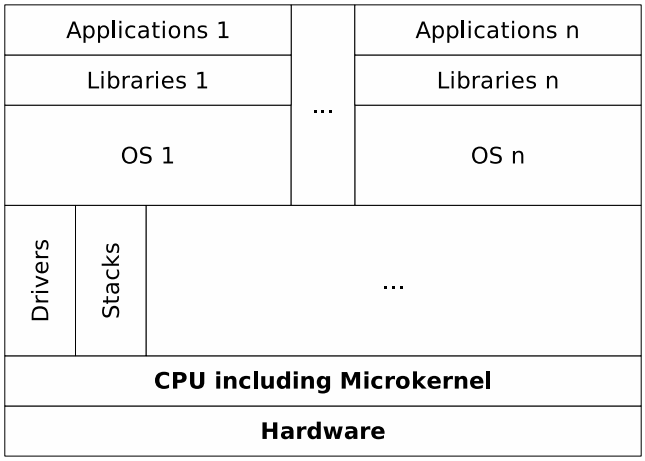

Figure 8: The μ-kernel is integrated into the processor's architecture

[2] Jochen Liedtke. On μ-kernel construction. *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[3] Jochen Liedtke. μ-kernels must and can be small. *5th Workshop on Object-Orientation in Operating Systems (IWOOOS)*, October 1996.

[4] Jochen Liedtke. Achieved ipc performance (still foundation for extensibility). *6th Workshop on Hot Topics in Operating Systems (HotOS)*, May 1997.

[5] William Stallings. Operating systems. internals and design principles. 3rd (international) edition. 1998.

[6] Lok Sun Nelson Tam. A comparison of l4 and k42 on powerpc. *The university of New South Wales*, Dec 2003.