CALU 1999
Linux on L4

Andrew O'Brien
andrewo@cse.unsw.edu.au

Conrad Parker
conradp@cse.unsw.edu.au

`http://www.cse.unsw.edu.au./~andrewo/l4linux/`

Version 0.2, 29th June, 1999

Contents

1 Introduction

The L4 microkernel is a lean, efficient microkernel capable of running Linux with performance comparable to native implementations. This offers several features that are not otherwise available, such as the ability to run additional L4-based operating systems on the same hardware and at the same time.

L4 is designed around fast message based synchronous IPC. It has a simple external paging mechanism and domain based security. As such it only provides a minimal set of abstractions, enough to support a fully fledged operating system but no more.

The effort required to port L4 itself to new hardware is roughly the same as the effort required to port Linux to the same hardware; the size of an L4 implementation is about the same as the architecture dependent portion of a Linux kernel. However, once L4 is ported to a platform, the same microkernel can be used to support other operating systems running on that platform, such as Mungi, DROPS and possibly the GNU HURD. L4 was originally written for Intel i386 hardware, and has since been ported to MIPS and Alpha, as well as reimplemented on Intel.

The functionality given by L4 is strictly defined. Hence, once a port of L4 has been made, little work is required to ensure its maintenance. We have been running the same version of L4 at UNSW for over a year, whilst extensively developing L4-based operating systems, with only two or three bugfixes and no major modifications to L4.

L4 has been ported to run on x86 (twice), alphas and MIPS. The MIPS implementation of L4 is released under the GPL.

As with a port to any other platform, porting Linux to the L4 microkernel requires the implementation of low-level kernel features such as interrupt service routines, driver support, signalling and the system call mechanism. Unlike hardware ports however, the L4 port can be implemented in C. The design of the resulting system is of course slightly different to that of a standard hardware port.

This presentation briefly introduces the concepts underlying the L4 microkernel, and then concentrates on the design of Linux on L4.

2 What is a Microkernel ?

2.1 General

As the name suggests, a microkernel is a very small kernel – it is a piece of software which implements a small subset of the functionality of a traditional operating system kernel. μ-kernels serve two purposes: they abstract away the particulars of the hardware, and they provide a stylised set of services to the systems built on them. A good μ-kernel implements functions thought to be useful to a wide range of larger systems, from specialised embedded controllers to fully-fledged multiuser operating systems.

The research into pure μ-kernels has slowed over the past few years due to the fact that in the past, in what could be called the first-generation of μ-kernels (late 80s/early 90s), the implementations have proved too slow to support any real operating system using the μ-kernel as a base.

Many first generation "micro" kernels evolved from monolithic approaches and so this was indeed the case, but second generation μ-kernels like L4 and QNX have shown that they can overcome these problems and be a viable base for extension.

2.2 Design Philosophy

Make it small.

Covered in more detail in sections 3 and 4.2.

2.3 Why Use Microkernels ?

Microkernels are good for experimentation, teaching and research into operating systems principles. For example, at UNSW we have a one semester subject in which students implement their own operating system with demand paging, a simple filesystem, process control and terminal I/O, which would be hard to achieve in so little time without the use of a μ-kernel.

The layer of abstraction between the hardware and the operating system allows some form of mutiplexing of operating system implementations. See figure 1 on page 3

We don't need to redesign architecture interface with each new OS. Of course, this goal is of little value in an environment such as pure Linux development.

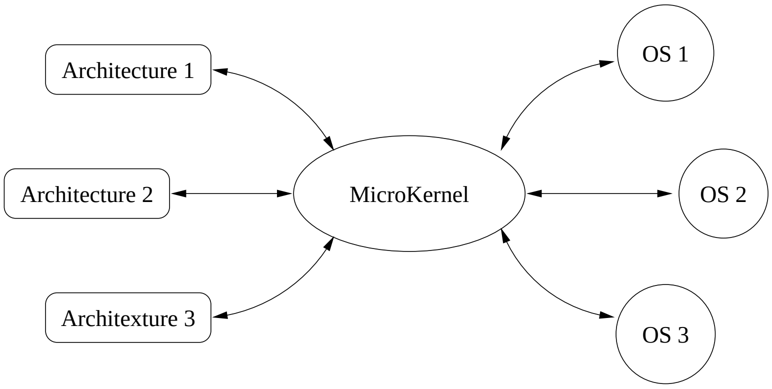

Figure 1: Design Multiplexing

2.4 Why Not Use Microkernels ?

However, these are not the criteria used for creating a production operating system. The μ-kernel adds a layer of abstraction between the operating system's services and the hardware, which not surprisingly is a small performance hit.

3 Microkernel Comparison

3.1 L4 vs Mach

Quick comparison of L4 and Mach

- functions (L4 simpler) 7 syscalls
- size (L4 smaller, compare lines of code to architecture dependant part of Linux kernel
- speed (benchmarks?)
- programming interface (MiG etc yeah!)

3.2 Others: QNX, Amoeba, Chorus, AmigaOS

There are many message-passing microkernels used for operating system research with systems such as Amoeba and Chorus which hadly ever see the light of day.

By far the two most popular message passing systems are QNX and AmigaOS Classic (ie versions up to and including AmingaOS 3.5). Interestingly, the next generation of AmigaOS (5.0 onwards) will be based on QNX.

***More info here ? (I know I can't be bothered)

4 Introduction to L4

4.1 History

The L4 microkernel was originally designed by Jochen Liedtke of Dresden University of Technology in 1995(6?****) as an imaginitively named successor to Leidtke's L3 microkernel. Shortly afterward, Liedtke went to work for IBM in the operating systems research division at Watson. Liedtke's implementation of L4, in pure i386 assembly, was not freely available and it was basically unsupported while he worked at IBM. As a result of this situation Kevin Elphinstone, a Phd student at UNSW, implemented L4 on a custom MIPS4x00 board loosely based on the Algorithmics P4000i.

Some time later in the Dept. Computer Science Dresden University of Technology Germany work was begun on porting L4 to the Alpha. They then reimplemented L4 on i386 in C++ (mostly just to show it could be done - they are quite aware of the performance difference and also quite unconcerned). The port to alpha is nearing completion due to collaboration between UNSW and Dresden and a local port to the StrongArm is in the works.

4.2 L4 Design Philosophy

The most fundamental task of an operating system is to provide a mechanism so as to share resources securely. This is really the only reason why there needs to be an operating system at all.

By definition, a μ-kernel is to be as small as possible, therefore the main design goal behind a μ-kernel must be minimality without compromising security.

To quote from the L4 user manual:

> A service or feature is to be included in the μ-kernel *if and only if* it is impossible to provide that service outside the kernel without loss of security.

So when you think of services that a normal OS kernel provides internally because of security concerns - eg DMA, does this necessarily need to be the case ? No. The driver needs to be trusted, certainly, but if there is sufficient security to guarantee a restriction on the priviledged operations that they can carry out while in user space then this is what needs to be done.

L4 is a lean kernel featuring fast message based synchronous IPC, a simple external paging mechanism and a security mechanism based on security domains. It only provides a very minimal set of abstractions.

4.3 L4 Basics

The L4 μ-kernel *needs* to provide:

- Threads
- Address Spaces
- Inter-process communication
- Unique Identifiers for context-free addressing in IPC operations

Address spaces Contain all the data, other than hardware registers, that can be accessed by a thread operating within this given space. It is just a set of mappings from virtual to physical memory, as you would expect.

One major idea of L4 is to support recursive constructions of address spaces by user level servers outside the kernel. The initial address space sigma-zero (σ^0) essentially represents the physical memory. Further address spaces are created by **granting**, **mapping** and **unmapping** logical pages (of size 2^n) which can range in size from a single page to an entire address space.

An owner can either "map" or "grant" sections of its address space to other address spaces.

map The owner still retains unmapping rights over that section of memory. Therefore the mapper can revoke mappings from its mappees at any time. This is fully hierarchical.

grant The owner relinquishes all control over that address space and no longer has a valid mapping for that region. The grantee inherits full control over this region - although the parents of the granter may still theoretically revoke that area at any time as it was originally mapped by them.

These are all just operations on page tables - no data is actually copied.

This is secure, in terms of the 3 operations above as it is only operating on virtual memory. Obviously it is not secure in the sense that when a page is mapped to another address space the owner has no control over where it is then mapped, but this is outside the scope of the service provided by the μ-kernel.

This method of user-controlled paging allows great flexibility for servers to implement many different memory mangement policies.

IO ports are treated as address spaces - mapped in and out as needed.

Threads The basic execution abstraction. A thread is an activity executing within an address space. It contains a UID, a register set (including an instruction pointer and a stack pointer) and an exeption handler.

Threads are extremely lightweight and cheap to create/destroy. This with fast IPC is the key to efficient μ-kernel operation.

A **task** is just a set of threads sharing the same address space. Within L4 there is a constant number of tasks - marked either active or inactive. Inactive tasks can be donated from an owner to another task.

IPC Cross-address space communication or Inter-process Communication (IPC) is one of the most fundamental μ-kernel mechanisms and we obviously want this to be as fast as possible given that all communication is being done either directly or indirectly with μ-kernel IPC.

This brings us to the concept of Clans and Chiefs.

Chief A tasks creator.

Clan The tasks directly created by a chief

All messages go via chiefs to other clans in a fully tree like in operation. This allows chiefs to directly interfere in the message passing of their "children". As all resource allocation is done through IPC this also gives us another layer of security for user tasks.

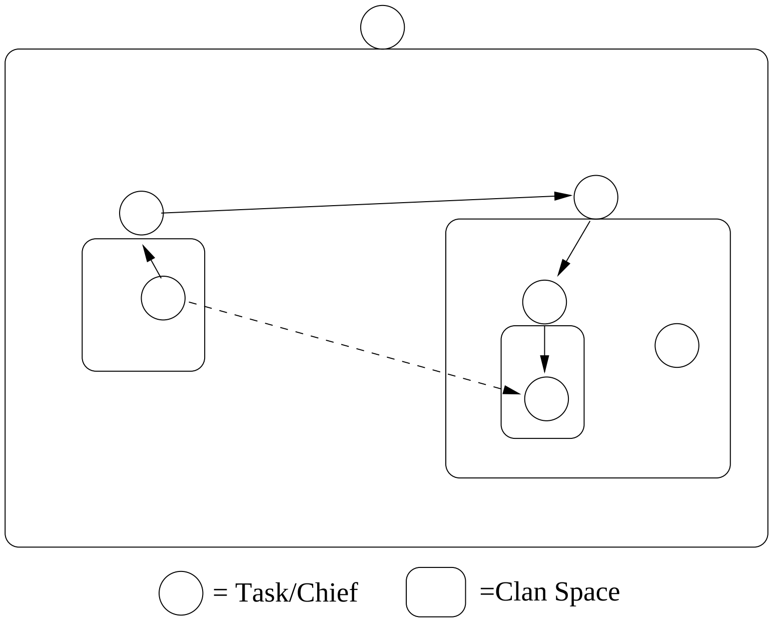

Figure 2: IPC Message Path in L4

4.4 L4 Resource Allocation

L4 only provides the tools and the associated security for allocating resources. It operates on a first come first served allocation strategy.

This is not necessarily bad in this case as the servers are the only initial tasks and therefore have the chance to grab everything they need before the user tasks kick in. Also, as discussed above, all access to resources is through IPC so the clans and chiefs structure can also stop unauthorised requests.

Hardware interrupts are transformed by the μ-kernel into IPC which is then sent to the appropriate thread. This forms the basis for implementing all device drivers outside the kernel.

4.5 Who Uses It ?

UNSW, Dresden, IBM ** Fix

5 How Is An OS Built on a Microkernel ?

Andrew Tanenbaum on:

> Microkernel vs Monolithic System
>
> Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in 'kernel mode.' This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.
>
> The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel. They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O. Examples of this design are the RC4000, Amoeba, Chorus, Mach, and the not-yet-released Windows/NT.
>
> While I could go into a long story here about the relative merits of the two designs, suffice it to say that among the people who actually design operating systems, the debate is essentially over. Microkernels have won. The only real argument for monolithic systems was performance, and there is now enough evidence showing that microkernel systems can be just as fast as monolithic systems (e.g., Rick Rashid has published papers comparing Mach 3.0 to monolithic systems) that it is now all over but the shoutin'.

5.1 Multiserver Design

To get an idea of how a multiserver operating system works, we'll take a look at the GNU Hurd.

```
:~/src/hurd/hurd-19990329$ ls -F
BUGS               config.sub*        libftpconn/        nfsd/
COPYING            configure.in       libhurdbugaddr/    pfinet/
ChangeLog          daemons/           libihash/          pflocal/
INSTALL            debian/            libiohelp/         proc/
INSTALL-cross      defpager/          libmom/            release/
Makeconf           doc/               libnetfs/          serverboot/
Makefile           exec/              libpager/          storeio/
NEWS               ext2fs/            libpipe/           sutils/
README             fstests/           libports/          tasks
TODO               ftpfs/             libps/             term/
aclocal.m4         hostmux/           libshouldbeinlibc/ trans/
auth/              hurd/              libstore/          ufs/
benchmarks/        hurd.boot          libthreads/        ufs-fsck/
boot/              include/           libtreefs/         ufs-utils/
```

```
bsdfsck/          init/             libtrivfs/        usermux/
build.mk.in       install-sh        login/            utils/
build.mkcf.in     isofs/            mkbootfs/         version.h.in
config/           libdirmgt/        mkinstalldirs*
config.guess*     libdiskfs/        mount/
config.make.in    libfshelp/        nfs/
```

Most of the directories here are services which are implemented directly as servers.

```
:~/src/hurd/hurd-19990329$ ls ext2fs
ChangeLog     devio.c      ext2fs.c     ialloc.c     pokel.c
Makefile      dir.c        ext2fs.h     inode.c      sblock.words
balloc.c      ext2_fs.h    getblk.c     msg.c        storeinfo.c
bitmap.c      ext2_fs_i.h  hyper.c      pager.c      truncate.c
```

You can see that each server can then have its own specialised pager etc.

This style of design forces independence of servers, are ability to change services on the fly.

This is the traditional "proper" way to build a microkernel based OS. The theory is, that by cleanly separating independent services and making maximal use of multithreading such a design, well implemented, could outperform a monolithic OS.

However we are yet to see this in practice and we'll take that statement with a grain of salt thiiiiiiis big until we do.

5.1.1 Why This vs Monolithic Approach ?

Linux has shown that it is possible to cleanly separate the development of many services and multiple architectures whilst still building a monolithic kernel.

5.1.2 Then How the Hell Does Linux Work ?

- development of each service is generally done by different people, hence the code evolves to independence
- architectural differences are, where possible, hidden behind such functions as `inb/outb`, `copy_\{to,from\}_user`

5.1.3 Problems With This Approach

- process separation of kernel and user space. How solved in microkernels ? (TiMEOUTS)

- stack tracing, killing of processes and killing back IPC connections. Consequence of different *use* of process model.

  On the multi-server implementation stuff, along with the stack killing problem:

  Linus, in reply to ast:

  > ¿ A multithreaded file system is only a performance hack.
  > Not true. It's a performance hack /on a microkernel/, but it's an automatic feature when you write a monolithic kernel - one area where microkernels don't work too well (as I pointed out in my personal mail to ast). When writing a unix the "obsolete" way, you automatically get a multithreaded kernel: every process does it's own job, and you don't have to make ugly things like message queues to make it work efficiently.

5.2 Monolithic Server Approach

5.2.1 Why This vs Multiserver Design ?

This is not the ideal solution but:

- it does take advantage of hardware abstraction
- it doesn't have to clean up message queues when processes die.

5.2.2 Why Is L4Linux Done This Way ?

The most immediate reason for this design is to be able to make as much use as possible of the Linux codebase. Indeed, as Linux is not designed as a multi-server kernel, rewriting it as one would have involved re-writing most of Linux.

6 Porting Linux or L4Linux In More Detail

6.1 Intro To Porting Issues

This project inspired by the x86-L4 developers in the Dresden Operating Systems Group at Dresden University successfully porting Linux to the x86 platform and showing that stability and speed were indeed possible and quite achievable within the framework of a monolithic kernel running as a server on top of L4. Much of their experiences will prove useful in terms of knowing exactly which parts of the Linux kernel code on which to concentrate, but then again, the the implementation issues involved with a MIPS4600 are a little different than that of an x86 box ...

Linux, as I mentioned, is designed with portability in mind and the kernel is written so as to provide an abstracted interface to the low level operations. This means that the amount of code that must be rewritten at a low level is minimised.

Actually, much of the code in the Linux kernel is written in such an abstracted manner.

6.2 Architecture Dependent vs Independent sections of the Linux kernel

Architecture Independent

- Process / Resource Management
- Filesystems
- Networking subsytems
- Device Drivers (note that these are still hardware dependent, but provided a similar enough hardware base, can be reused without change)

= about 98% of the code in a full kernel distribution

Architecture Independent Except for changing a few device drivers, porting linux should only require changes to the architecture dependent part of the system. This encapsulates the underlying hardware architecture and provides support for

- Interrupt service routines
- Low level driver support (eg DMA)
- Methods for interation with user processes

Also must implement

- Switching between kernel contexts
- Copyin/copyout for transferring data between kernel and user address spaces

- Signalling
- Mapping and unmapping mechanisms for address spacess
- The linux system call mechanism.

From the users perspective, this defines the kernel's Application Binary Interface (ABI).

For managing address spaces, Linux implements a 3 level architecture-independent page table scheme. By defining appropriate macros it is possible to implement this on the lower levels with (relatively) minor hassle.

Interrupt handlers are divided into Top halves and bottom halves. Top halves run at the highest priority, directly triggered by hardware interrupts and can interrupt each other. The bottom halves run at the next lowest priority and can only be interrupted by other top halves, not bottom halves or the Kernel.

6.3 General Porting Design

In this project, we are aiming for full binary compatibility for the Linux server.

We shall be taking a single server approach - single Linux server on top of L4.

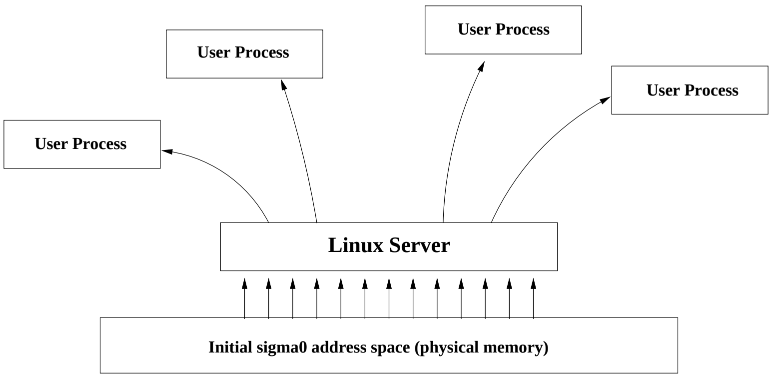

Figure 3: Linux single-server approach on L4

Linux maps physical memory one-to-one into the kernel's address space. That is, σ^0 maps everything it has spare into the address space of the kernel which then acts as pager for all user tasks.

At this point the page table implementation not yet defined.

So as not to modify the Linux kernel more than necessary, only a single L4 thread is used to in the server for handling all activities induced by syscalls and page faults. To avoid blocking we obviously need to multiplex this thread.

The Linux kernel uses interrupt disabling for synchronisation and critical sections. As we can also achieve this at an L4 user task level, this can be used with little modification.

6.4 Specific Porting Designs

6.5 Interrupt handling and device drivers

As L4 translates interrupts into IPC, we simply implement the top half handlers as threads waiting for such a message, one thread per interrupt.

Interrupt handler thread:
do
wait for interrupt (L4 IPC)
top half interrupt handler()
od

Another thread executes all bottom halves once the pending top halves have been completed. To avoid concurrent execution of the linux server and the interrupt threads, we execute the interrupt threads and the bottom half thread at a higher priority than the kernel.

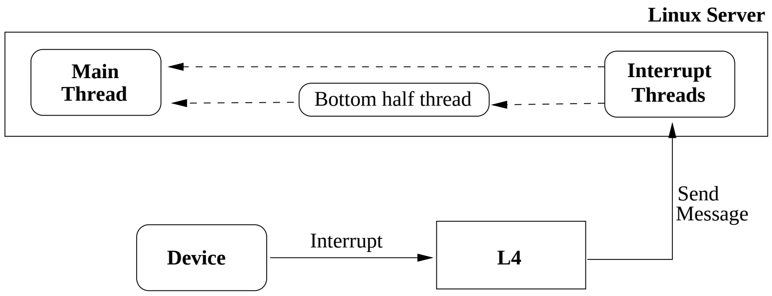

Figure 4: Linux interrupt implementation on L4

6.5.1 Process Model

Each user process will be implemented as an L4 task - ie a set of threads within the same address space. The Linux server creates these threads and registers itself as their pager. All page faults are then converted by L4 to RPC to the Linux Server which then handles the respective map/unmap operation.

The server maps the emulation library and the signal thread code into an otherwise unused pages in the high-end of the address space for each process. The emulation thread just handles communications with the server - does not implement unix functionality on its own.

6.5.2 Syscalls

A syscall is an IPC between a user process and the server. We have the following options:

1. A modified libc.so which uses L4 ipc to call linux server.
2. A modified libc.a as above
3. A user-level exception handler (trampoline) which emulates the native system call trap instruction by calling a corresponding routine in the modified shared livrary.

The library options are faster whereas option 3 gives true binary compatibility.

6.5.3 Signalling

Normally linux delivers signals to user tasks by manipulating their stack, stack pointer and instruction pointer. For security reasons L4 restricts such interthread manipulations to threads sharing the same address space. Therefore we add a signal handler thread to each user process.

The signal thread gets a signal from the server, it then causes the main thread to save its state and enter Linux by manipulating the main threads stack pointer and instruction pointer.

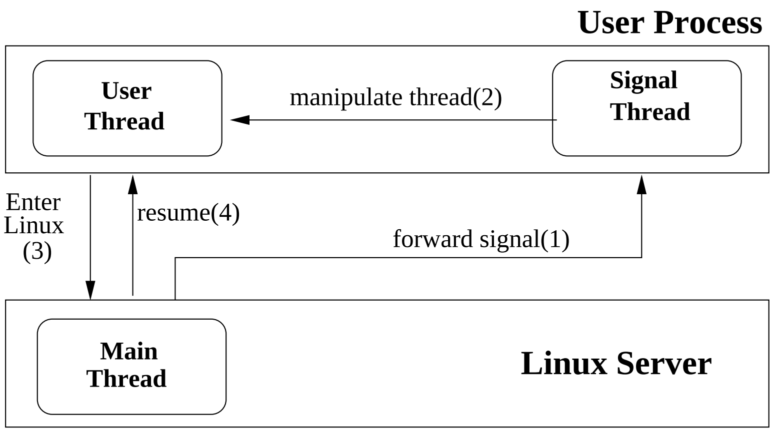

Figure 5: Linux signal implementation on L4

Note - the signal thread and emulation library are not protected against the main thread. But then again there is no real benifit in modifying this to the user process. Global effects of signalling (eg killing) are implemented by then server - the signal thread only modifies the user process.

7 Conclusion