Linux: a Portable Operating System

Linus Torvalds

Helsinki January 31, 1997

Master of Science Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Linus Torvalds

Työn nimi — Arbetets titel — Title

Linux: a Portable Operating System

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master of Science thesis	January 1997	52 p.

Tiivistelmä — Referat — Abstract

We explore the hardware portability issues in Linux that were uncovered when porting the operating system to multiple CPU and bus architectures. We also discuss software interface portability issues, especially with regard to binary compatibility with other operating systems that can share the same hardware platform. The approach taken in Linux is described, with a few example architectures covered in some more detail.

Computing Reviews Classification:
D.4 (Operating Systems), C.1 (Processor Architectures), D.2.7 (Software Engineering — Portability), D.4.7 (Operating Systems — Organization and Design)

Avainsanat — Nyckelord — Keywords
Linux, Operating System, Portability, Alpha, Sparc

Säilytyspaikka — Förvaringsställe — Where deposited
Department of Computer Science Library, serial number C–1997–12

Muita tietoja — Övriga uppgifter — Additional information

Acknowledgements

While the Linux project has been closely associated with me personally, partly due to the name, I would like to make it very clear that the Linux operating system is a huge project done co-operatively by lots of people all over the world.

Even if you discount all the user-level programs that are an integral part of any running Linux system, just the kernel contains code from hundreds of people from all around the world.

Thanks to all of you.

Contents

1 Introduction

This thesis is an introduction to the portability issues of the Linux kernel. It is not only meant to be my Masters thesis, I hope it will actually be useful for people who want to port Linux to other architectures, or for people who just want to know the design issues and goals of Linux on different platforms (as opposed to the more general goals of Linux in general).

In an attempt to make this practical, I try to explain the general issues, usually followed by a comment on one or more specific ports with some pointers on how something is implemented in practice[1].

Note that while the term "Linux" is often used to denote the whole *system* that is built up from the basic kernel and all the system and user applications that are usually running on top of the kernel, in the context of this thesis only the kernel itself is considered. The portability of system and user applications is here considered a totally different issue, even though there are obviously some common concerns.

Also, you should not expect this to be a line-by-line (or even very low-level) explanation of the kernel — rather the opposite. Instead of trying to explain what all the specific functions do, this paper tries to explain the basic ideas, and the specific examples should be seen as just clarifying how Linux works on some specific hardware in some specific case.

The thesis generally follows a common format: each section first explains the generic problems and potential solutions, to then be followed by an "Implementation" section that gives Linux-specific implementation details. The reader should be able to follow the text even with little or no knowledge of Linux itself, although a general knowledge of the issues is obviously expected.

History of Linux portability

People who have followed Linux from the very beginning may find the title of this paper, "Linux: a Portable Operating System", a rather ironic statement. Being portable was not what Linux was about initially; the early versions of Linux were extremely unportable.

[1] For the same reason this paper will also be available in an electronic version which I will try to keep up-to-date as people send me comments or as changes to the system occur. Linux is by no means a static system, and some of the issues, especially when it comes to multiple CPU's, are so young that major changes can still happen.

The original Linux was not only extremely PC-centric, it *wallowed* in features available on PC's and was totally unconcerned with most portability issues other than at a user level. The original `Intel 80386` architecture that Linux was written for is perhaps *the* example of current CISC design, and has high-level support for features other current CPU's would not even dream about implementing (see for example [CG87]).

Linux did not even try to avoid using the `x86` features available to those early versions — quite the opposite. Linux started out as a project to find out exactly what you could do with the CPU, and as such it used just about every feature of the CPU you could find ranging from using segments for inter-process protection to hardware assisted process switching.

However, the initial unportable approach was a case of an unportable *implementation* rather than an inherently unportable *design*. The goal of being compatible with other UNIX's resulted in a system that had portable interfaces despite the implementation details. That portable design essentially made Linux itself reasonably portable.

The first Linux that was based on a architecture different from the Intel 80386 was the port of Linux to the Motorola 680x0 family that actually got started rather early in the development of Linux. However, the original Linux/68k project was not really concerned with portability, but rather with making Linux run on 68k-based Amiga and Atari computers.

Please note here the fundamental difference between the concepts of "portability" and "ability to run on different architectures". A portable program is designed and implemented in such a way that it will run on different systems, while even a non-portable program can be *forced* to run on other systems. The original Linux/68k project did not try to make Linux portable, it only tried to re-write the Linux kernel so that the new version would run on a new architecture instead of the original one.

While the Linux/68k project was in itself a huge step forward, the real portability work began when the author was offered an Alpha system by Digital in the hope of making Linux work on the new Alpha architecture. Very early it became clear that in order to be able to maintain both the stable Intel-based platform and support a new and in some respects radically different platform the kernel really needed some major re-engineering to make it fundamentally more portable. The issues and the end result is what is described in this paper.

The current situation is that the Alpha port has been totally successful, and is in wide use and fully integrated into the main kernel sources. At the same time, the

portability issues have not only been considered from the viewpoint of the Alpha port, but on a more general scale. As a result, other porting efforts are much easier, and the original Linux/68k project has updated its base to the new portable layout and an integration of that work is in progress [Law97].

In addition to the Alpha and 68k ports, the port to the Sparc architecture has also been very successful and is also in active day-to-day use and is mostly integrated into the base sources. In addition to these four platforms there are also ports to various other architectures, including the MIPS [Bäc96], PowerPC [Dou96] and the ARM (and StrongARM) [Kin96] lines of CPU's.

At the same time the Linux kernel has also been ported to virtual environments like the Mach [App96] and L4 microkernels [Hoh96]. In those environments the Linux kernel is running on top of another kernel that provides another view of the physical machine, to potentially get the best of both worlds.

Portability concerns

Unlike some other software projects, an operating system does not have any inherent reason for existing: the role of an operating system is only to be the interface between user programs who do the real work and the hardware it is to be done on. An OS without viable user programs is useless, and an OS that does not work with the hardware available is likewise pointless. The worth of an OS is measured in how well it supports user processes, and how well it can implement the services it provides on the hardware.

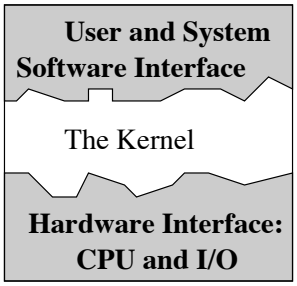

Figure 1: The Kernel: the Interface between user and hardware

As such, the kernel itself is defined by *both* the hardware it runs on *and* the software it supports (Figure 1). When writing the OS and when porting it to a new architecture, both parts of the equation should be taken into account. The person doing the port seldom gets to choose his or her own idealized hardware, and similarly the porting effort may be constrained by user level software requirements.

On the hardware side, the concerns can be roughly divided into two categories: the architecture of the CPU subsystem (including memory management, caches and optionally multiprocessing details) and the architecture of the I/O interfaces. On the other hand, the software side problems are generally

with regard to compatibility: to what extent (if any) do we want to be compatible with other, possibly similar, operating systems.

However, more important than the hardware and software porting details themselves is having a general design that makes it easy to be portable. It should allow for easy addition of new interfaces (both on the software and the hardware side) without impacting existing code. In the case of Linux, this is especially important due to the way development is done: a distributed development community with no direct face-to-face contact between developers and little central authority to resolve conflicts.

While the different porting efforts should be able to work without impacting each other, general maintenance concerns require extensive sharing of code. If the different architectures end up doing everything on their own with very little common code, the whole goal of portability has been lost: the different platforms might as well be considered separate projects.

For maintenance reasons the architecture-specific code needs to be small and simple, and all the complex functionality should be handled by architecture-independent code that is shared across all platforms. This, in turn, requires a common concept shared across all architectures. That concept is called the virtual machine.

The virtual machine concept

The most common approach to creating a portable system that has to be able to adapt to different hardware and software requirements is to use the concept of a *virtual machine* [GC94, p. 70], [Tan92, p. 22].

Instead of having separate architectures that all would have to be handled separately, the idea of the virtual machine is to create an idealized abstraction of the machine, and through that abstraction let the common code operate on the real hardware. The common code need not itself be aware of the actual hardware details, and as such can operate on any physical hardware as in Figure 2.

The idea of a virtualized machine is not new in itself: just the fact that the operating system is generally written in some high-level language is in itself a virtualization of the underlying hardware with the help of the compiler that does the translation from the high-level language to the hardware-specific machine code.

With the virtual machine abstraction, we only extend the virtualization already offered by the high-level language to something that can handle the details needed to implement the kernel on top of the actual physical hardware. While most high-level

languages are good for virtualizing normal user level activity, we need to extend that to cover all the details of the hardware.

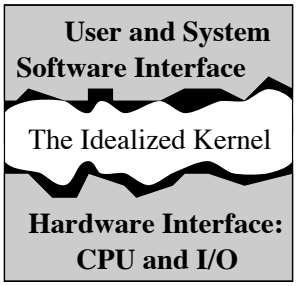

Figure 2: The Virtual Machine

The concept is in wide use because it is so useful: on Windows NT the virtual machine is called HAL (*Hardware Abstraction Layer*), while SVR4 UNIX has a more limited HAT (*Hardware Address Translation*) layer to handle just the machine-dependent virtual memory address translations [GC94, pp. 92].

With this kind of idealized kernel that does not interface directly to the hardware but to the virtual machine, the act of porting the operating system to another platform or another user level binary interface is no longer a matter of rewriting everything. Instead, only the outlying interfaces to the virtual machine have to be implemented on top of the physical machine, and the majority of code can be left untouched.

Obviously, the choice of virtual machine semantics is of primary importance to how easily the system can be ported, and how efficient the resulting operating system is. Implementing a virtual machine that is at too high a level results in much architecture-specific code, while a virtual machine that specifies too much may not map efficiently onto some specific hardware.

The Linux kernel virtual machine

While the concept of a virtual machine to handle portability issues cleanly is very useful and widely used, it does add a layer of abstraction to the kernel. In fact, depending on how this is done it can add *two* layers of abstraction: the layer between the kernel and user space, and the layer between the kernel and the hardware.

In many cases, the layer between the kernel and user space can be ignored, because the raw kernel interfaces are often exported more-or-less directly as system calls to the user processes. This is the historical approach taken by most operating systems, simply because the operating system was made for the hardware, and no previous standard for the user level interface existed.

However, in the case of Linux, previous user level standards often exist on architectures Linux is ported to, and to be optimally useful Linux needs to conform to those standards in order to leverage off existing applications. Thus, both the hardware and the user-level abstraction layers are generally required.

The problem with abstraction layers is that they add overhead when concepts have to be translated from layer to layer. This has generally been one reason hardware abstraction layers have been shunned: they add extra processing that could be avoided by programming directly on the bare hardware or software interface. This has to be taken into account, especially as one of the goals of Linux in the first place was to be efficient.

2 Linux — the Design and Implementation

Unlike most modern research operating systems, the Linux approach to operating systems is very pragmatic, and rather than concentrating on the *design* of a new kernel the focal point has been on a solid and efficient *implementation*. Instead of throwing away proven concepts they have been refined in new ways. [Tor93]

The Linux design is based on three main design issues that have directly influenced the implementation. The main issues are:

- Simplicity. The obviousness of this sometimes means that it is overlooked, but an operating system kernel is a complex entity that has to be able to work in an uncontrolled and potentially even hostile environment. Programming errors in the operating system are much less acceptable than in most normal programs, and the security issues are paramount. A complex design is harder to verify against either errors or security issues, so simplicity of the basic services is required.

- Efficiency. The kernel is involved with almost all activity in the machine, and as such the kernel must be efficient enough to *never* be seen as a performance constraint.

- Compatibility. While the basic operations of a kernel are of supreme interest to researchers in the operating system area, most people do not want to know what is going on as long as their programs work. As such, one of the most important features of an operating system is the lack of surprises it offers to the user, be he a normal end-user or a programmer. Even new features should be offered as a *superset*, rather than *instead* of functionality that the user is accustomed to.

These design issues, coupled with a very pragmatic approach to programming, has led to a system that shares features of both the traditional monolithic kernel design and of the newer research projects into modern microkernels. The microkernel design itself was discarded here due to doubts about the efficiency and simplicity of the approach, but during the development many of the features usually associated with microkernels have been implemented on top of a more traditional monolithic kernel.

The design has also been strongly influenced by the general availability of Linux source code to anybody with access to the Internet. That has not only led to a very

dynamic development community that is physically spread all over the world, it has also directly impacted the design itself.

Rather than concentrate on a concept of a kernel binary that interfaces to the rest of the world (be it hardware or software), the Linux approach has been to have sources available that can be used to create the binary we want. This approach allows the actual interfaces to be specified at compile time rather than at run-time.

The more traditional link-time and run-time interfaces are naturally also available, and give access to more dynamic feature sets. By using run-time dynamic linking of modules into the kernel, the user can add and remove features as needed, but the availability of sources always adds the possibility of increased performance by specifying the configuration at compile time and letting the compiler handle interfacing issues.

2.1 The design — compatibility

The three main design issues – simplicity, efficiency and compatibility – have naturally formed the basis for the kernel implementation. The compatibility issue essentially defines the external shape of the system while the issues of simplicity and efficiency define the internal implementation of it.

The wish to be compatible with other UNIX-type operating systems more or less defines the kernel interface that is exported to system and user applications. Note that compatibility extends to outside the machine itself: not only does the compatibility issue require a compatible programming and user interface to the kernel, it also involves the integration of the system with other external requirements. These requirements include following standard protocols on external networks, but also being able to live together on the same machine with other operating systems. This can involve supporting a common disk partitioning scheme, or a particular boot-up sequence.

As being compatible so clearly restricts the external interfaces of the system, many new operating system projects choose to be less compatible in order to be able to fully express the wishes of the designer. Instead of being constrained by what others have done, such a project can soar to new heights of design beauty, at least in the eyes of the designer.

Sadly, those projects invariably tend to fail in real life, getting only a niche market (if that) due to their lack of common applications or interfaces. As mentioned

earlier, the worth of an OS lies not in what the OS can do, but in what the OS allows *others* to do. Therefore a more pragmatic approach is often required.

While compatibility clearly restricts the design of the system, it at the same time makes the system easier to integrate with others, and also potentially allows a large base of existing applications to be used directly on top of the new kernel. In the case of Linux, being compatible with generic UNIX meant that a plethora of freely available UNIX applications were directly useful on top of Linux. So while the collar of compatibility may feel chafing at times, it also allows the system to be built on top of the work done by others.

However, while being compatible is good for the usability of the system, it tends also to be boring. A grey mass of compatible operating systems is not what makes the heart of system designers beat more quickly, nor does it allow the systems to show much innovation other than at an implementation layer.

Because the Linux project has been done non-commercially by people all over the world connected by the Internet, a boring system would simply not work: lacking most of the money-related incentives Linux depends on being vital and interesting to attract developers. As a result, Linux inevitably has accumulated interesting extensions to the basic UNIX compatibility, and one of the challenges has been to add these extensions in a manner that cleanly integrates with other code.

2.2 Implementation

While the basic shape of the system is constrained by the compatibility issues, the designer is free to decide how that shape is in fact *implemented*.

While most modern operating system research tends to favor a microkernel and object oriented design, Linux has leaned towards a much more traditional design. Like the UNIX that it tries to be compatible with, Linux uses a very traditional monolithic kernel mostly written in portable ANSI C.

Even though portable C is generally well suited for systems programming (that was one of the original uses of the language, after all), the Linux kernel additionally uses some features offered by the GNU C compiler that make it easier to implement certain features efficiently. The two notable extensions to C used are inline functions and inline assembly language code embedded within C.

In addition to the code written in C, some low-level routines have been written directly in assembly language either due to performance reasons or due to circumstances where the C semantics are unable to cover the exact code generation

requirements (usually requirements imposed by the hardware itself, such as fault handling routines).

The traditional design was chosen because of serious doubts about the performance and simplicity of microkernel-based operating systems. While much work has been done on the performance front of microkernels (see for example [BGG+91]), the very fact that such extensive work is needed in the first place indicates that the issues are by no means obvious.

Also, while simplicity has been used as one of the arguments *for* a microkernel-based design, that argument seems dubious at best considering the complex interactions required between the different modules in a microkernel design. This matter is generally made worse by the complex performance issues, and the basic simplicity of microkernels is often overwhelmed by the complexity of these issues.

While this thesis by no means tries to argue against microkernels per se, the author feels that the advantages of microkernels have been overstated in the literature, and that a simple and straightforward traditional approach may serve the needs of users better. Microkernels tend to be best suited for distributed systems rather than traditional UNIX type operating conditions. Even then the question remains at what level the distribution should be done[2].

That said, the very basic UNIX design to some degree favors a microkernel approach to user level services. Even a traditional monolithic UNIX kernel can be seen as only a small microkernel when considering all the services done by fundamental non-kernel UNIX daemons. While many other operating systems tend to handle user verification and session handling in the kernel (and in some cases even graphics services!), that mentality is against the very philosophy of UNIX, where those services are provided through user-level daemons.

2.3 Kernel organization

The basic Linux kernel is directly organized around the primary services it provides: process handling, memory management, file system management, network access and the drivers for the hardware. These areas correspond to the kernel source directories `kernel`, `mm`, `fs`, `net` and `drivers` respectively. Additionally, these areas are often sub-divided into specific services (see Figure 3).

The basic portability tenet in Linux is the *single source* concept, i.e. all architectures share the same basic source tree. The architecture-dependent code is

[2]See Chapter 4.4.2 on Linux implementations of distributed environments

available in the `arch` directory which has sub-directories for each architecture supported. While there are no absolute rules about what an architecture has to offer, common concerns have generally resulted in similar organizations of the architecture sub-directory, and most of them follow a setup with separate directories for process handling (`kernel`), memory management (`mm`) and library routines optimized for the specific architecture (`lib`).

This hierarchical subdivision of the sources not only makes it easier to get an overview of the system, it also acts to modularize the kernel. While nothing prevents code to be written that ignores this conceptual layout, maintainability requires that the developers keep clean interfaces between different parts of the system. The modularization not only keeps the sources more understandable, it is also a requirement when many developers are working on the same project at the same time. Different developers can work on their specific area of the kernel without generally having to worry about what happens in other areas.

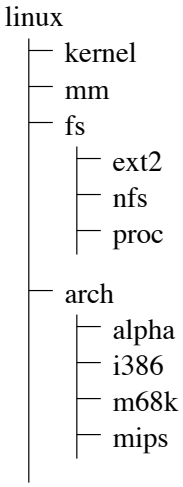

Figure 3: Kernel organization

The modular nature of the kernel has also led to a limited kind of object oriented design. While the C language used by the kernel does not inherently support this kind of object orientation, the effects can be similar through the use of function pointers and common data structures. This is especially notable in the file system code, where the virtual file system layer that implements the general UNIX file system semantics will see the specific low-level implementations through this kind of object interface[3].

2.4 The virtual machine implementation

While abstraction layers required by a virtual machine approach can generally be a performance problem, Linux has a big advantage: source availability and a monolithic kernel.

Rather than having a binary-level abstraction layer, the abstraction layer can be moved up at a source level. That way many of the transformations required can be handled by the compiler at kernel

[3]The use of C++ would have allowed these kinds of object interface features to be described on a language level. This was tried early in the Linux development process, but at that time the compiler technology was not stable enough, and there were concerns that a too high-level interface might hide details that are critical in a kernel.

compile-time rather than at run-time, and will as such not impact performance adversely.

As it is a monolithic kernel, there are no external entities and fixed protocols that have to be taken into account, so the compiler is free to optimize away operations that are not needed on specific architectures. More specifically, because nothing else depends on the internal data structures those data structures can be tailored at a source level to match the user or hardware requirements as closely as necessary.

The way the Linux kernel virtual machine is implemented is through the creation of architecture-specific header files and a common API between the generic kernel code and the architecture-specific files. These header files create the image of the virtual machine that the common code is running on.

The virtual machine header files define the machine by using the C language preprocessor (`#define` directives) and by using various type defines and inline functions to implement the virtual machine on top of the physical hardware. While inline functions are not part of standard C, they are implemented by most `C` compilers, and the support for them by the normal Linux `C` compiler (GNU `C` compiler, also known as "gcc") is extensive. As the GNU `C` compiler is itself very portable, the use of `C` extensions is not a problem.

The use of gcc also allows the architecture header files to use inline assembly routines to implement code that cannot be expressed efficiently with portable C. This, together with inline functions, allows the compiler to efficiently map the virtual machine semantics on top of any specific hardware, and means that the virtual machine abstraction results in little or no performance loss.

In addition to the architecture-specific header files, each architecture has its own subdirectory in the `arch` directory in the system sources. This directory contains the set of rules to build the kernel sources for each architecture; what files to use, what special options the compiler needs for this architecture and so on. This approach allows a maximum of flexibility with regard to architecture-specific code, and does not impose any restrictions on what the code does.

Even in the architecture-specific parts of the kernel, common concerns have resulted in a basic layout of the architecture-specific kernel sources. Most architectures tend to have the subdirectories `kernel`, `mm` and `lib` available. These subdirectories contain code for generic kernel functionality (e.g., system call interface, trap handling), low-level memory management code (e.g., page table initialization and page fault handling) and optimized architecture-specific library code (e.g., memory copying, network checksum calculations) respectively.

Additionally, many architectures have software floating point math emulation for hardware that does not have floating point available, or for hardware which needs software support for special cases like underflow and overflow situations.

3 Software Interface Portability Issues

One area of concern with regard to operating system portability that is often overlooked is the software interface side. The issue has traditionally been ignored, because the operating system designer was also in charge of designing the interface to the system and application software, and as such the operating system kernel could freely implement any interface it wanted.

However, the computer industry has matured, and most modern operating systems have to live with the fact that interface standards already exist, and cannot be ignored. Sometimes the standards are on a source level, allowing much freedom in actually implementing the standard. In the UNIX world, the most well-known example of this is the POSIX (*Portable Operating System Interface*) standard [Ins96].

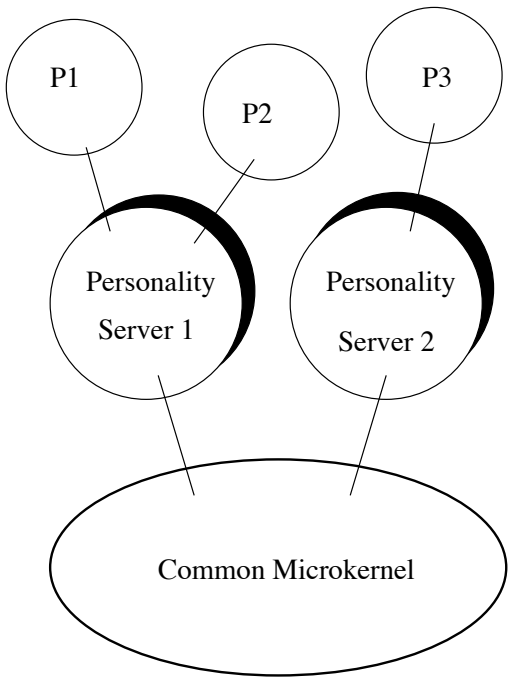

Figure 4: Multiple personalities

More often, the standard is a binary compatibility standard that is required in order to run legacy applications designed for a previous operating system. In this case, the designer must very closely follow the exact interface details of the binary standard so that no legacy applications break when the system is upgraded to a new operating system kernel.

If no previous binary standard exists, the operating system designer is generally free to create any binary interface to the kernel, and in that sense a lack of a standard can be very liberating. However, even then the designer is generally limited by the hardware protection mechanisms in selecting the exact interface for the kernel, and as such the architecture to some degree always limits the interaction between user mode and the kernel.

Another issue that is sometimes faced is the existence of *multiple* standards on the same platform, with subtly different[4] and possibly even contradictory requirements. While this is fortunately rare, it may be one of the primary concerns for the system designers.

The issue of multiple interfaces to the operating system is usually handled by having so-called OS *personalities*. One personality handles the interfacing issues of one interface standard, and all personalities share the same common core operating system routines.

In one sense, the concept of OS personality is very similar to the issue of portability to the hardware, and the same virtual machine abstraction can be used to handle the case of multiple interfaces. The idealized kernel does not directly implement *any* of the interfaces, and the software portability issue is then a matter of mapping the wanted user interface on top of the idealized kernel image.

One large difference between OS personalities and hardware platforms is generally that the kernel only runs on one physical platform at a time, yet it potentially has to handle multiple personalities concurrently. Microkernel operating systems generally handle this by the addition of a *personality server* for each personality (see Figure 4).

In this kind of personality server approach, the overhead of such personality handling is quite high, though, and is one of the primary reasons for work on migrating threads (see for example [FL94]) and moving system services into the kernel address space (see [LHFL93]). More importantly, while the concept of a personality server allows a personality-independent view of the system, most real system usage tends to favor one *primary* personality. The other personalities are then mainly used for backwards compatibility or emulation of non-native systems.

3.1 The implementation of the software interface

Because the primary goal of Linux was never to act as a platform for emulating other systems, the Linux approach to the software interface personality problems is straightforward.

Rather than supporting the notion of multiple personalities of equal importance, the Linux kernel interfaces tend to have just one primary personality for which the system is optimized. Having a primary personality allows the use of compile-time optimizations for the common case, and means that the internal structures can be

[4]Sometimes the differences are not so very subtle.

optimized for the expected use of the system. The integration between personalities and the generic code is much tighter (Figure 5).

Such a tight binding between the common code and the primary software interface does not preclude the use of secondary personalities, and should be seen not as a limitation but an optimization. Also, it should be noted that the primary personality can be adapted to the platform and is not inherent to the virtual kernel itself. On the Alpha platform, for example, the primary personality is the Digital UNIX compatible interface, while Linux on the Sparc has a primary personality that looks like the original Sun operating systems on the same hardware.

In addition to the primary personality, any number of secondary personalities can exist. These secondary personalities can be dynamically loaded into the kernel at run-time as needed, as shown by the iBCS2-personality[5] on the original Intel platform [Int91].

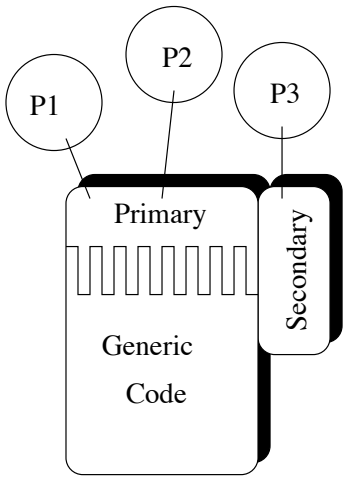

Figure 5: Primary personality

Note that these secondary personalities have the same access to all kernel resources as the primary personality, and the difference is not so much a conceptual one as a question of optimization: for which personality has the internal data structures been optimized. While the primary personality generally needs to do only minimal translation from internal kernel data structures for user mode requests, secondary personalities may have to translate user requests into a format suitable for the kernel.

In addition to the secondary personalities, there can also be *tertiary* personalities: environments that are not directly supported by the operating system kernel, but that can be emulated in user space, often with at least limited kernel support. This is how Linux/x86 is able to run DOS and Windows programs — through an external emulator and minimal support from the kernel to expose the hardware features needed for efficient emulation.

Similar emulators exist for running Macintosh, Commodore 64 and other binary formats. Another example of this is the ability to run Java programs through a Java interpreter or just-in-time (JIT) compiler [GM95]. In the case of Java (and

[5]iBCS2 stands for *intel Binary Compatibility Specification 2*, and is the binary format generally used by non-Linux UNIX-like operating systems on the i386 platform.

potentially other binary formats), the kernel will automatically notice a Java binary and start up the appropriate emulation technology for seamless integration with native programs.

3.2 Personalities

In the design of Linux, the basic approach has been to adhere to the standard UNIX semantics as closely as possible. However, as many UNIX haters will tell you, the concept of *standard UNIX* is to some degree a contradiction in terms.

One of the problems with UNIX is that while the basic approach to how things work is the same, different vendors have done versions of UNIX that differ in the details. And through the quarter century history of UNIX, quite a lot of differences have evolved.

While the UNIX community has started to address this very issue through various standards, and while the current state of UNIX is by no means as splintered as it was only five years ago, the situation is by no means perfect. As some form of UNIX is available for just about any computer hardware in existence, the UNIX standards obviously tend to be source standards rather than binary interface standards.

In addition to varied hardware, various vendors have their own value-added features that they have specialized in and that they use to differentiate themselves inside the basic UNIX community. That, together with 25 years worth of backwards compatibility for legacy applications tends to make the whole term "UNIX" a very fuzzy issue.

In order to be really compatible with what people perceive as "UNIX", it is thus not enough just to follow the standards. The standards are generally much more limited than any real UNIX is, and while getting POSIX-branded has been a favorite pastime of many operating systems (not all of them even remotely UNIX-like) the fact that they follow just one standard by no means indicates that the system behaves as the *user* expects it to behave.

As such, the Linux compatibility issues go far beyond just the paper standards, and rather than be a subset of all UNIXes the idea is to be a *superset* of the behavior people expect.

While being a true superset of all UNIX behavior is not technically feasible (or necessarily even a good idea), Linux tries to distill the basic essence of UNIX, and implement all the relevant features people expect from such a system. Adherence

to standards generally make it easy to port programs that have been written with the standards in mind, but Linux goes further than that.

Now, being a superset of what people expect should in theory be simple. Sadly, that is not the case. The main problem is that while the basic UNIX operations are similar, the difference in details can often result in two systems doing the same thing in different and contradictory ways.

For example, while the standard interface for reading the contents of a directory is through the `readdir()` interface on a C library level, the actual implementation of that interface at a lower level can be very different on different UNIX operating systems. And at a binary compatibility level Linux would need to implement *all* of those different interfaces.

Considering at the same time that the current default Linux sources contain no less that 17 (sic!) different file systems, the duplication of code would be horrendous if all these file systems would have to know about all the possible interfaces to read a directory. Obviously the act of reading the directory contents has to be handled with a virtual interface that can do the appropriate translations for all the different systems.

Another problem in compatibility is the different layouts of data structures and the different sizes of various operating system types that different UNIX operating systems support. This indicates that the basic kernel routines should be type-independent, in order to be able to handle different layouts. While this typing can be done at compile time (allowing for optimal code to be produced for the primary personality), the sources should still be able to accommodate any type layout without becoming a mess of conditional compilation.

Finally, the issue of various environment constants has to be handled. Error numbers change from system to system, as do the command numbers used by various system calls (notably `ioctl()` and `fcntl()`). All this has to be cleanly separated into architecture-specific header files that allow the kernel to share the basic sources yet differ in details.

4 Hardware Portability Issues

While software interface portability issues are important for any operating system, it is generally the *hardware* portability issues that most people tend to consider first when talking about portability. These hardware portability issues can roughly be divided into a few main areas.

The first concern, and an area that is also seen in normal code outside the kernel, is the issue of the data representation of the CPU. Most of the CPU internal issues are hidden by the compiler: the programmer needs generally not worry about the number of registers of the CPU or the actual code generation. But other issues often need to be taken into account: the byte order and alignment of values in memory and the width of internal registers, for example.

Most other portability concerns are problems that generally do not impact user-level programs: differences in the memory management and caches in a system, and possibly multiprocessing details. Similarly, the differences in I/O architectures should never be seen by the user, but they directly impact the operating system.

4.1 Data representation portability issues

An obvious problem with any portability project is the CPU architecture of the hardware being ported to. In fact, so obvious is this problem that other problems like the I/O architecture are often overlooked completely.

Much of the CPU portability is handled by using a portable language and having a compiler that is able to translate the code to various different architectures. That kind of portability is more or less required by any large project, and the *real* portability work is then in making sure that semantics that are not guaranteed by the language will port correctly across different platforms.

With a complete enough language the compiler could handle all the issues in CPU portability, and porting would be a matter of just recompiling everything to the new architecture. However, currently no such language exists that is suited for systems-level programming like creating an operating system.

Linux, like most operating systems these days, is written in C, which is reasonably portable, while allowing a close interaction with the architecture. That close interaction allows the programmer to program on a very low level if required, but the very fact that `C` allows that kind of low-level programming obviously also exposes the machine more than some other programming languages. As such, badly

written `C` code by no means automatically works the way you would expect on another machine.

During the porting of Linux to other architectures, there have also been situations where a compiler error had to be fixed before work could proceed. This was true mainly on the Linux port to the Alpha, because the gcc port of Alpha was itself reasonably new when the Linux port was started.

4.1.1 Different data sizes

One of the more obvious differences between different CPU's is the size of the internal data registers and the virtual address space. Thus, for example, the Intel 80x86 has 32-bit registers and address space, while the registers and address space on an Alpha are 64 bits wide. This results in the standard data types in `C` being different on different architectures.

This difference in the basic data types is often something that is brought up as the main problem in portability. However, at least in the case of the Linux kernel itself, the size of the data types were mostly irrelevant, and while there was much nervousness about moving from a 32-bit architecture to a 64-bit architecture like the Alpha, it turned out not to be much of a problem in real life.

In normal portable code, the size of the underlying hardware data types generally does not matter. In many cases it is totally irrelevant whether a "`long`" in `C` is 32 bits or 64, and most of the kernel code will use whatever size that happens to be the native size for some particular machine.

However, in some cases data size does matter, usually because the size is defined by some external entity — this is usually the case in file systems where the data layout on the physical disk has a certain machine-independent standard associated with it, for example. Similarly, most of the networking code depends on being able to work with addresses of a certain size, and hardware interfaces to device controllers generally have a very rigid type structure independently of the hosting CPU type.

Data sizes — implementation

In Linux, the architecture-specific header files export a small set of specific types to be used when a piece of the operating system needs to use some specific type-size. Including the file `<asm/types.h>` makes the `C` types `u8`, `u16` and `u32` accessible when something specifically needs a 8-, 16- or 32-bit unsigned integer.

The corresponding signed data types (`s8`, `s16` and `s32`) are also available, but use of them should be minimized as any object that needs to have a certain number of bits is generally not a normal integer.

Due to standard `C` name space rules, these types are only available when compiling kernel code. If a specific type needs to be exposed to a user-space implementation, there exists versions of the same types with two underscores prepended to the type name (`__u8`, `__u16` etc). These are meant to be used mainly in public header files where the normal names for these types cannot be exposed.

Unresolved issues

While the current Linux kernel has proved to be portable to a wide variety of hardware, including different word sizes and endianness, the issue of data type portability is by no means completely handled.

One data size problem that is often overlooked is the size of the character set. While an 8-bit character is currently the standard, it totally ignores the issues of wide characters used mainly in Asia. The Linux kernel sidesteps the issue completely by not doing any character translation in the kernel itself, and leaving all such issues to user space.

As far as the kernel is concerned, all data is a stream of 8-bit bytes, and the interpretation of those bytes (possibly by combining two or more bytes into a wider character) is left to the user programs. The standard console driver supports various character translations through the use of UNICODE and loadable font-sets, but the policy of setting up the fonts and translation tables is handled at a user level. The tools to do this both under a graphical user environment and on the system console already exist, so in that sense the issues are already solved.

Another type-related issue is the problem of machines with word-sizes that are not powers of two or machines that are not byte-addressable. While those kinds of machines have existed in the past and are still in use in places, it is considered unlikely that such machines will be created in the foreseeable future (at least for a larger market where a Linux port would be interesting).

For similar reasons the problems with integer bit-level representation has not been considered, and it is likely that machines that implement other integer representations than two's complement would require some portability work. The current portability efforts have not been worried about such issues.

4.1.2 The impact of alignment restrictions and byte order

Much more complex than the size of the data is the memory access details of data on some specific CPU. Even when two CPU's share the same data types, the representation of that data in memory can be different and many CPU's restrict memory accesses to only aligned addresses.

For example, on the original Linux platform, the Intel 80x86, the CPU will allow a 32-bit word to be loaded and stored at any memory address regardless of alignment. The CPU is byte-addressable and little-endian, i.e. it stores the least significant byte of a multi-byte entity at the low address and the more significant bytes at higher addresses in memory (Figure 7).

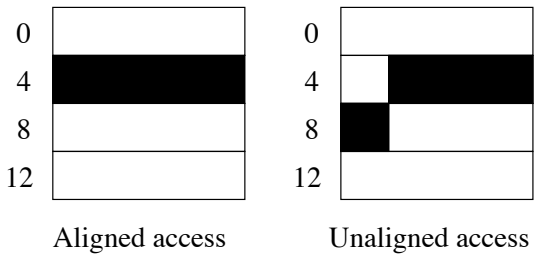

Figure 6: Alignment

In contrast, the Sun Sparc line of computers are big-endian and memory accesses must be aligned: if a program tries to store a 32-bit word at a byte address that is not evenly divisible by four the CPU will trap (Figure 6). Most other modern architectures are also likely to trap on unaligned accesses.

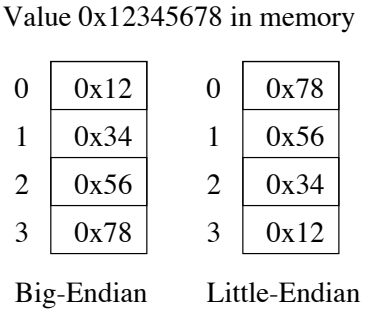

Figure 7: Endianness

These kinds of memory access details turned out to be much more of a problem than the size of the data itself. The size of registers generally does not matter for most code, and the cases in which it *does* matter are well-defined and generally not problematic to find.

With alignment, the problem is that while the compiler can make sure that all internal kernel data is correctly aligned, the kernel cannot assume alignment for external events, notably user space argument pointers.

For byte order, the problem is that the effects of byte-order are generally much more prevalent and harder to find than just the size of data. The size of any data structure can often easily be changed by just changing the type of the `C` structure that is used by the kernel to access the data. However, when it comes to byte order there is no equivalent change we can make on the `C` language level, so any byte

order changes have to be handled by hand by the programmer rather than let the compiler sort it out.

Alignment and byte order — implementation

As with type sizes, both alignment restrictions and byte order issues have their own header files that allow the kernel virtual machine to use any alignment or byte order. The virtual machine is assumed to always trap on unaligned memory accesses, and thus if an access is known to be unaligned at compile time it can be handled by a simple inline function that does the extra work to load an unaligned datum as two separate aligned loads[6].

However, the alignment of many accesses cannot be known at compile time. To be safe, all these accesses would need to be done with the potentially complex unaligned access function, and that would impact performance negatively especially in cases where unaligned accesses are rare. This is especially true with many user level accesses, where the user-supplied pointer is almost always aligned, but can potentially be unaligned in some special cases.

To handle those kinds of rarely unaligned cases efficiently, the kernel virtual machine assumes accesses are aligned and if an unaligned trap occurs, the trap handler will fix up the occasional unaligned case. Trapping will be much more expensive than handling the unaligned cases explicitly, but if the unaligned accesses are rare this works out well. To make sure the user is aware of the potential performance degradation the system will print out a warning when an unaligned fault has to be handled. Under normal circumstances these messages are never seen.

Note that some code, notably the networking code, will actively try to align all the data structures it uses but will not *guarantee* that the structures are always aligned. This is a prime example of code where the optimistic unaligned trap handling works efficiently.

When it comes to byte order, the virtual machine makes no assumptions about the native byte order of the machine. Indeed, in most cases byte order simply does not matter. However, in those cases where byte order matters (on-disk layout with a certain specified byte order, or networking code, for example), the functions exist to convert the native CPU byte order to and from both little-endian and big-endian[7].

[6] On hardware where alignment does not matter, the inline function obviously needs not do any extra work, so there is no overhead for architectures that handle unaligned accesses in hardware.

[7] Little-endian byte order is used mainly by various file systems and device drivers that use the standard PC byte order, while big-endian byte order is used by the Internet networking protocols.

Unresolved issues

Especially when it comes to alignment all the issues are still not resolved completely. While handling the uncommon unaligned cases through trapping is efficient and guarantees the correct behavior even in the presence of potentially unaligned data, this particular design decision may have to be re-thought.

The problem with the current virtual machine design is that while it maps very efficiently onto most hardware, there exists various hardware implementations that do not conform to the hardware model expected by this virtual machine design. Most notably, the design *requires* that the hardware supports unaligned accesses through trapping, and that may not always be the case.

While most hardware conforms to the expectations of this virtual machine model, there are cases where the CPU may handle unaligned memory accesses by simply ignoring the low address bits rather than trapping on the access. This is one example of how choosing the wrong virtual machine concept can result in problems that may require a redesign in the future.

The other unresolved issue with regard to unaligned data is the state of the compiler support for the concept of unaligned accesses. While the current virtual machine depends on explicit programmer action to support unaligned accesses, a future version might depend on compiler directives to let the compiler automatically create the necessary code to handle the unaligned accesses. Such compiler support already exists to some degree, and can be used hide the issue from the programmer the same way the issues of type size have been hidden.

4.2 Kernel memory management

One of the most complex areas in the kernel in general, and with regard to portability in particular, is the virtual memory management.

Memory is one of the most fundamental resources in the system, and as such the performance of the memory management layer is critical to the system. Making memory management efficient is thus of primary importance: not only do the routines have to be fast, they have to be clever too, sharing physical pages aggressively in order to get the most out of a system.

The memory management code is also fraught with race conditions and deadlocks, and trying to share as much of the complex memory management routines as possible would be a good thing in order to not have to maintain multiple separate copies of this complexity.

However, to make matters even worse, memory management is typically one of the areas where there are absolutely no hardware standards, and different CPU's use very different means of mapping virtual addresses into physical memory pages. As such, memory management is one area where traditionally most of the code has been very architecture-dependent, and only very little high-level code has been shared across architectures even though we would like to share a lot more.

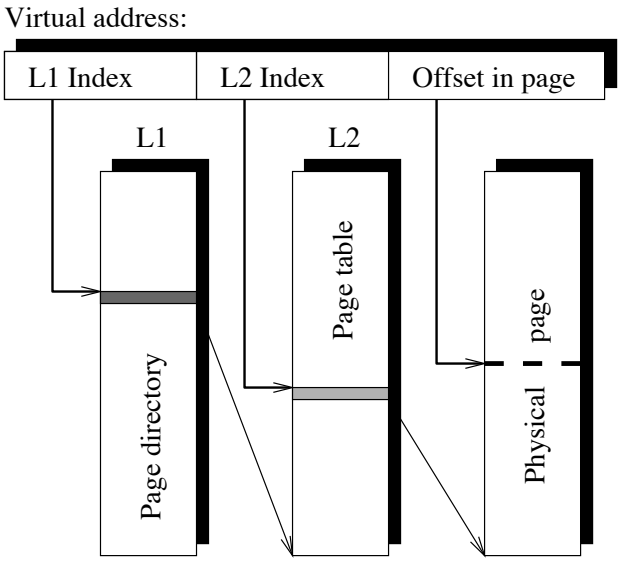

Figure 8: Two-level page table tree

While the generic virtual to physical memory mapping can be seen as any function that maps a virtual address into a physical address, extreme performance requirements mean that the function has to be reasonably simple. In fact, on a low level all current CPU's use a on-chip virtual memory mapping cache usually called a TLB (*Translation Lookaside Buffer*), and in the end all virtual memory mapping schemes translate into filling this cache with the appropriate translation information.

While any mapping strategy is possible, current CPU's tend to handle the virtual memory translations in three different ways: with page table trees, with hash tables or with a pure software-fill TLB. But even when the basic approach is similar in two architectures, the low-level details are often very different.

Depending on the memory management unit, the page tables may contain extra information aside from the necessary protection and translation information. Some architectures support special *dirty* and *accessed* bits, to be used by the VM routines to determine whether a user has written to a page or not, or whether the page has been recently accessed. Other architectures expect the operating system to keep track of this information by hand.

For example, the original Linux platform, the Intel 80x86, has a two-level page table tree (see Figure 8), and implements both dirty and accessed bits in hardware. In contrast, while the Digital Alpha from a system software viewpoint also has a normal page table tree, on the Alpha the depth of the tree is three due to the larger

virtual address space, and the Alpha lacks hardware support for dirty and accessed bits.

On the other hand the PowerPC and some of the Sparc CPU's do not have a real page table at all: they have a hash table that is used to look up the physical page that corresponds to a virtual address (Figure 9). If the page cannot be found in the hash table they trap to software. [Int93b]

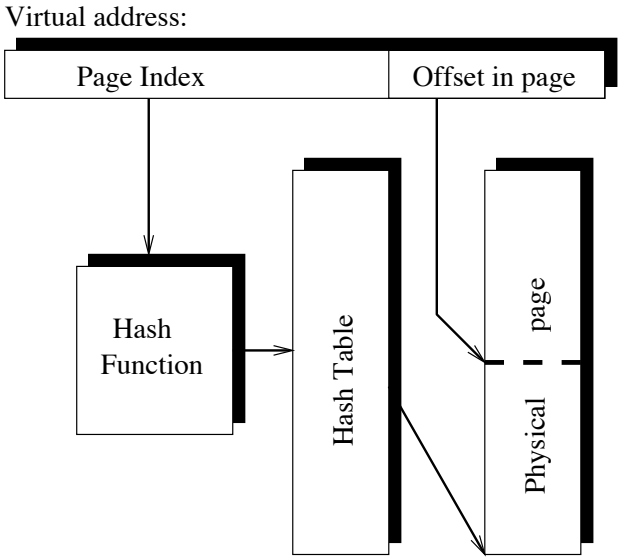

Figure 9: Hashed in-memory TLB extension

Because the hashed memory mappings generally cannot fully describe the virtual memory setup, they are more appropriately called *in-memory extensions to the on-chip TLB* so as not to confuse them with full page tables.

Finally, MIPS CPU's and the newest UltraSparcs from Sun do not have any architecture-specified page tables at all, they only have the on-chip TLB and any miss in the TLB will result in a software trap to refill the TLB[8] (or handle a page fault if no physical page is available for the offending virtual address). [Int93a]

4.2.1 Memory management through virtual page tables

As seen above, no standard way of mapping virtual memory exists. Indeed, the Sparc line of CPU's have used all three different mapping strategies described above in different versions of the architecture. And yet, despite these fundamental differences in physical hardware we would like to use as much common code as possible.

The way this is accomplished is by having a common virtual mapping scheme in the Linux kernel virtual machine (see chapter 1), and mapping that common memory management scheme onto the physical hardware. This allows us to share all memory management code over all supported architectures, and any improvements

[8]This is also the case with the Digital Alpha architecture, but the Alpha architecture also specifies a low-level software layer called *PAL-code* that makes it appear as if the hardware had three-level page table [Dig92, pp. 3–2].

to the memory management are automatically supported on all platforms. The only thing that the architecture-specific code needs to know about is the mapping from the virtual machine onto the physical hardware.

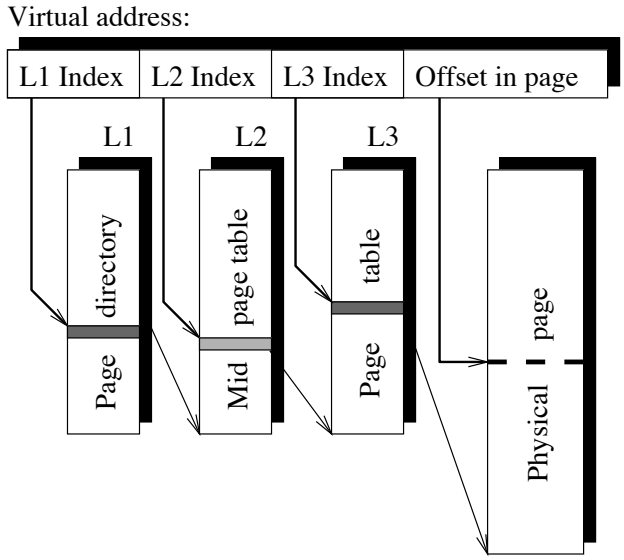

Figure 10: Three-level page table tree

While the principle of the virtual machine approach is simple to grasp, the details are not as obvious. What mapping scheme should be used in order to make the translation to the hardware as efficient as possible, yet be generic enough that the scheme is useful as a superset of any realistic real hardware? If the virtual machine is too limited, it cannot take advantage of large address spaces or special hardware features.

There are also secondary concerns: the virtual machine memory mappings must be memory-efficient, so that the mapping information does not take up a lot of physical memory that could be used to better advantage for file system caching or running user programs. Remember that not only does the kernel have to keep the page tables of the virtual machine in memory, the page tables of the *physical* machine also take up space.

With all the requirements placed on the page tables of the virtual machine, the choice in the end is not difficult. It turns out that a multi-level page table tree is an approach that can easily be expanded to match large virtual memory spaces by just adding levels. It is flexible, simple, and reasonably efficient.

Not only is a multi-level page table a good generic answer to the page table problems, it can also often easily be made to map closely to the actual hardware, so that the mapping of the page tables from the virtual machine to the physical machine is easy to do. In fact, by choosing the right virtual machine page table setup, the same page tables can be used by *both* the kernel virtual machine *and* the physical memory management unit. In those cases the mapping overhead is obviously zero.

Architectures which natively use multi-level page tables include the Intel 80x86, the Digital Alpha and the Motorola 68k family. The Intel hardware uses two-level page tables [CG87, pp. 465], while the 68k and the Alpha use three-level page tables [Dig92, pp. 3–1]. As three levels is enough to map on the order of 40–45 bits of virtual address space, and no current hardware supports more, that was the choice for the Linux kernel virtual machine (see Figure 10).

It is worth noting that a virtual machine page table tree of level n can easily be mapped to a physical page table of level $n-1$ by simply "hiding" one level inside another (Figure 11). This is actually used by Linux/x86 to create two-level physical page tables from the kernel virtual three-level page tables. The hiding approach also allows the future expansion of the kernel virtual page tables without breaking any current two- or three-level page table machines.

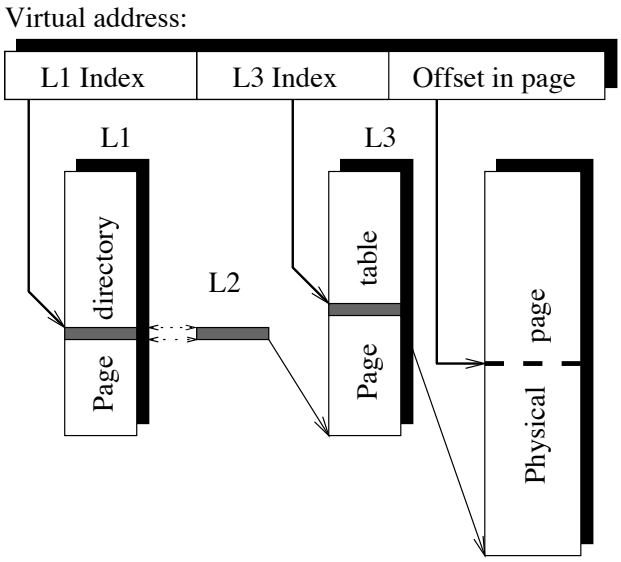

Figure 11: Hidden mid page table level

Mapping a multi-level page table tree onto a pure TLB architecture like the MIPS or the UltraSparc is similarly trivial. The TLB refill code can just follow the page tables by hand, and because of the simplicity of the data structures this refill code can generally be written in low-level assembly directly in the TLB miss trap handler for best performance.

The case of hashed in-memory TLB extensions like the PowerPC, the mapping is no longer as trivial, but there is a generic approach that works for these cases or any other memory management setup. The algorithm is as follows:

1. On-chip TLB misses.
2. Hardware does a hash table lookup, if mapping found: fill in TLB and restart the access.
3. Mapping not found: hardware invokes kernel page fault handler.

4. Kernel page fault handler does a kernel page table walk, and inserts the page found into the hash tables and returns.

5. The memory access is re-tried, and this time the mapping is found in the hash table in stage 2.

The above algorithm can clearly be used on any hardware, and is the fall-back position if a simpler mapping cannot be done directly.

4.2.2 Page table mapping coherence

While having separate virtual and physical page tables[9] is good for portability, they obviously also add a new problem: the problem of keeping the two page tables coherent. Some way of invalidating the hardware page tables when the virtual page tables have changed is obviously necessary.

Even when the virtual page tables map directly to the physical page tables and there is just one copy of the page tables itself, the same coherency issue makes itself felt in the form of TLB coherency. The internal CPU on-chip TLB generally still needs to be invalidated even though the external in-memory page table is updated correctly in synchronization with the virtual page tables.

Because the on-chip TLB invalidation is an issue regardless of the actual layout of the physical page tables, we can generalize the TLB invalidate concept to cover the case of a separate hardware page table as well. Thus, as far as the virtual kernel is concerned, *any* hardware page table is considered only an extension of the TLB. This concept allows us to create an algorithm for changing the virtual page tables as follows:

1. Change the virtual page tables.

2. Invalidate the CPU on-chip TLB information.

3. If the hardware page tables are separate from the virtual page tables, invalidate the hardware page tables.

Again, this algorithm will work correctly with any hardware page tables. The kernel needs to be careful about race conditions, making sure that the hardware page

[9]Note that even when the virtual and physical page tables can be mapped to use the same layout, they should be considered *conceptually* totally separate.

table information is never used after the virtual page tables have been changed, but as the changes to the page tables are well localized this is generally not a problem.

The main problem is to find the right level of invalidation granularity: invalidating too much hardware state can be very expensive especially with in-memory extensions of the TLB. This issue is closely linked with the issue of hardware support for multiple address spaces: when we change the virtual mapping of *one* process we should not necessarily have to invalidate all the hardware state associated with other running processes.

All of the information that describes the kernel virtual page tables can be found in the architecture-dependent header file `<asm/pgtable.h>`. This header file describes the actions needed to be taken when invalidating and modifying the virtual page tables, and is the main architecture-specific file in the whole memory management tree apart from the initial setup code and the low-level trap handling.

To give the reader some idea of the success of the virtual page table approach it could be noted that most of the current architectures can distill the required architecture-dependent page table information into a `<asm/pgtable.h>` file of only around 500 lines, much of which is comments or trivial one-line inline functions describing the hardware page tables.

4.3 Ensuring cache coherency and atomic operations

One of the issues that the original Linux for the Intel line of computers did not have to concern itself with was CPU cache coherency. Traditionally computers based on the `x86` line of CPUs have always done cache coherency in hardware, and neither the operating system nor the programs running on the CPU have needed to care about cache consistency.

However, the Intel CPUs are more the exception than the norm in this regard, and on most other architectures the operating system will have to be aware of the caches on the system, and make sure that coherency is maintained in software.

4.3.1 Instruction cache coherency

The simplest form of cache coherency problems arise from the common use of so-called *Harvard* architectures [HP96, p. 55], where the instruction and data caches are separate as in Figure 12.

While a separate instruction and data cache can result in less efficient cache use due to imbalances in the usage patterns for caches, it also allows the chip designer

to de-couple the instruction stream and the data stream, and thus allows for a simpler and more efficient implementation. Also, while a unified cache architecture can theoretically be more efficient in cache utilization, a Harvard style split cache does guarantee that the instruction cache does not get completely flushed by data accesses (and vice versa) and can as such work as a balancing factor.

However, while the introduction of separate instruction and data caches are generally beneficial for performance, the split caches also introduce the problem of coherency between the two separate caches. The issue of instruction cache coherency is a matter of how to keep the instruction cache up-to-date with respect to changes done to the data that it caches.

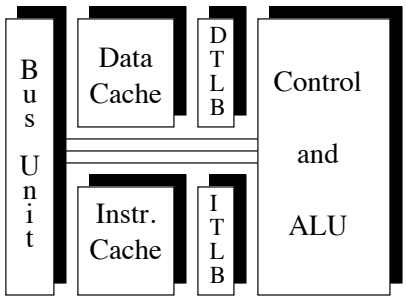

Figure 12: Harvard caches

The most obvious solution is to make sure that the hardware maintains cache coherency, and every time any memory data is changed the change is noted and any relevant instruction cache contents are updated. This is what the original Linux platform did, and is what Linux was originally designed for.

While hardware support for instruction cache coherency makes life simple for the programmer and was required for compatibility reasons on the Intel 80x86 platform, it is not the solution generally preferred by engineers. As a result, most other modern platforms do *not* support this kind of hardware coherency.

The main reason for not supporting hardware instruction cache coherency is that the reason for split instruction and data caches in the first place is to be able to de-couple the instruction stream logic from the data stream logic, and having to maintain coherency disturbs this de-coupling. In order to maintain coherency the instruction cache essentially has to snoop all data traffic anyway[10], and thus incurs more complexity than necessary.

More importantly, modern programming languages and paradigms eschew the use of so-called self-modifying code, and as such at least in theory it should never happen that the instruction stream is dynamically altered. The engineering standpoint on the issue thus boils down to the question whether to add complexity to support something that should never happen in the first place. The answer becomes self-evident.

[10]Or, more usually, the instruction and data caches have to maintain some exclusion policy that guarantees that the same data cannot be in both the data and the instruction cache.

Instruction caches under Linux

If modern programming languages and paradigms shun self-modifying code, why is the issue of instruction cache coherency a problem in the first place? Why cannot the operating system simply ignore the issue, and just refrain from using self-modifying code at all?

The reason for some form of instruction cache coherency becomes obvious after some thought. Even though the kernel need not modify *itself*, the kernel will obviously have to cause any running programs to be modified. Each time a new process is started and the image of the executable is loaded into memory the in-memory instruction image is modified, and some form of invalidation is required.

As such, every time a new program is run, the operating system has to support a limited kind of code modifications, even though it obviously no longer is a matter of *self*-modifying code. Similarly, each time the kernel changes the virtual mapping of a process it has to invalidate the old instruction caches if there is *any* possibility of the new mapping being tainted by old cached data that is no longer valid.

In addition to any mapping changes of a process, the normal way of implementing user-mode callbacks (*signals*, in UNIX terms) under Linux is by writing a short trampoline function on the stack of the process that is used for the return to kernel mode. While this is not the only possible implementation of user-mode callbacks, it is a very flexible one, and results in the kernel modifying the user process instruction stream and thus requires the instruction cache to be made coherent with the memory image.

Finally, while self-modifying code is frowned upon, no operating system should enforce its own set of morals upon the user programs unless that enforcement is required by security issues. As such, self-modifying code in user space should at least be possible, and may require kernel support. It should be noted that the kernel in general cannot be expected to know when processes modify their own instruction stream, and as such this support should not be automatic, but the user should have some way of requesting kernel help if the hardware does not in itself support any user-level interface to support instruction cache coherency.

However, despite all these concerns, the issue of instruction cache coherency is not overly problematic in the end. There are two reasons why instruction cache coherency can easily and efficiently be handled by software. The first reason is inherent in the way instruction caches work: the instruction cache never modifies the instruction stream. Because the instruction cache is never modified, maintain-

ing coherency is a simple matter of invalidating the current instruction cache and reading in the new modified copy. There are no conflicting write-back issues or any need for exclusive access to the instruction cache.

The other reason that instruction cache coherency is reasonably simple to handle is the fact that the circumstances under which an instruction cache invalidate is needed are well-defined and easy to pinpoint. Also, while an invalidate operation can be costly, the circumstances under which it is needed are so rare that it is very rarely a major performance issue.

Because the occasions where instruction cache invalidates are needed coincide with virtual memory management changes, the Linux kernel itself need never concern itself with the issues of flushing the instruction cache. If the architecture requires special instruction flush sequences, they can be hidden in the TLB invalidate code itself, and as such the kernel virtual machine can mostly ignore the issue.

Similarly, the case where instruction cache flushes may be needed due to user-level callbacks can be handled within the callback code itself, as that is by nature architecture-specific in any case. So while the architecture-specific code needs to be aware of the instruction cache issues, they do not need to be exported to the virtual machine level.

4.3.2 Data cache coherency

While instruction cache coherency can be reasonably easily handled, the issues of coherency within the data cache itself can often be very important. While the coherency issues with regard to instruction caches stemmed from the fact that they were de-coupled from the normal data cache, the coherency issues with data caches can be the result of two different circumstances: multiple users of the data and cache aliases. In both cases the same data exists in multiple places and thus data coherency issues arise.

Even though the coherency issues are basically the same as with separate instruction caches, the matter is much more complicated due to the fact that data caches are not static entities that do not change. As a result a simple invalidate of the stale data is no longer acceptable — the coherency issue is complicated by the need to have sane semantics for the case where there are multiple writers to the same location.

Additionally, while instruction caches and data caches are not supposed to overlap under normal circumstances, the same is by no means true when it comes to data

cache duplication. There are no guidelines that allow us to assume that data cache duplication would be an exceptional and rare occurrence, which further complicates the issue.

The normal case of data cache coherency problems arises from having two separate entities that both access the data. If one or more of the entities cache data for performance reasons, the fact that the data can exist in multiple places introduces the problem of coherency when the data is later modified.

It should be noted that having multiple entities that access the same data does not necessarily imply multiprocessing per se. A common concern even in uniprocessor environments is the cache coherency issues that are introduced by the use of I/O devices that use DMA (*Direct Memory Access*) to read or write the data directly from memory.

While most systems support cache coherency in hardware, there are again exceptions to the rule. The architectures that do not support hardware coherency require that any common data always be accessed through the main memory subsystem, and the operating system has to make sure that all caches are up-to-date and flushed to memory before any DMA activity occurs.

4.3.3 The case against virtual data caches

In contrast with the multiple user case, a cache *alias* occurs when the same location is found in multiple different areas in the *same* cache, leading to the same coherency issues as if there were multiple entities.

How do such cache aliases occur? A caching algorithm that places the same data in different locations can be considered an utterly broken caching algorithm, because not only does it result in the aliasing problem, it can also obviously result in bad cache utilization if the same area is cached over and over again.

While such a caching strategy can with good reason be considered stupid in the extreme, it is something that *does* exist in current hardware, and as such is something that the kernel needs to know about. The reason for these cache aliases is generally a virtually indexed data cache.

One of the most basic issues in caching is how the cache is *indexed*, i.e. how the cache lookup is done. The two main cache types are *virtually* and *physically* indexed caches. As the names imply, the difference is whether the virtual or the physical address of the cached data is used for cache indexing. [HP96, pp. 423]

In a physically indexed cache, the cached location is looked up using the physical address of the memory location, and as such one physical memory address will always be found through the same index and no aliasing can occur.

In contrast, a virtually indexed cache will use the virtual address as seen by the running program as the cache lookup index. This has some obvious benefits for CPU design: the physical address need not even be computed before the cache indexing can take place, and cache lookups can be made faster and can happen in parallel with the virtual to physical address translation.

However, the use of virtual caches directly lead to aliasing problems as different virtual addresses can point to the same physical memory location. As a result, the different virtual addresses can result in different cache locations being used to cache the same memory, resulting in the above-mentioned aliasing problem [IKWS92].

It should be noted that virtual indexing and the resulting aliasing is not a problem for instruction caches: while the aliasing may result in slightly lower utilization of the cache, this is often balanced by the simpler lookup, and because instruction caches do not modify their contents the worst effects of the aliasing problem never materialize. In fact, virtual instruction caches can often make the issue of instruction cache invalidation (see 4.3.1) simpler for the operating system.

However, in the case of data caches, the aliasing issues either require extra anti-aliasing hardware or it has to be handled by proper support by the operating system. As the reason for the virtual indexing in the first place was simplicity of hardware, it is generally left to the operating system to clean up the mess.

The most trivial example of a dual virtual mapping is the case when a page is available both in kernel space and in user space. As the kernel address space generally contains *all* physical memory, this happens with any user page. Happily, only in a few cases do these mappings actually conflict in the cache.

The reason that the double kernel-user mapping normally does not lead to aliasing problems is that in most cases a user process gets full control of any page it uses and the kernel never reads or writes to that page through the kernel address. So while the mapping exists both in user space and kernel space, this aliasing is not a problem. Care must be taken that the kernel always uses the user virtual address for any data copies, but as that is the normal mode of operation anyway the issue is moot (see later about memory management issues in Chapter 6).

Even though most common memory mappings do not result in aliasing problems, there are a few cases where this is not true, notably with memory mapped files. When a user process asks to map a file into user memory, the user-accessible page

is usually *also* accessed by the kernel through the page cache that is maintained for the mapped file.

As all user programs are loaded into user space as a memory mapping, this dual mapping of the same physical page onto multiple virtual addresses is extremely common. However, the overwhelming majority of all mappings are what in UNIX parlance is known as *private* mappings, i.e. the mappings are read-only mappings with regard to the common shared page, and if a process writes to the page, the virtual memory subsystem will force a trap and a new copy of the page to be allocated (so-called COW, or *copy-on-write* behavior).

With private mappings, there is still only one entity (the kernel) that can change the physical page associated with an aliased mapping, and UNIX memory mapping semantics do not require such private mappings to be kept 100% synchronized. The aliasing concern still requires that the kernel be careful the first time the kernel copy of the page has been read, but that is a simple matter of making sure that the data exists in main memory, and not only in the kernel virtual cache.

In short, for private mappings the data cache aliasing concerns with regard to cache coherency are not overly problematic. Because of the potential duplication of the same physical location in multiple cache locations, the virtual cache should still be larger than an equivalent physically indexed cache for the same performance [WHL93].

What makes virtually indexed data caches really bad for UNIX (and thus Linux) are the semantics of *shared* memory mappings which allow different processes to map in the same physical page in multiple locations, and allow full read and write accesses to the page. This is mainly used for fast sharing of data between applications, and is used by the X server with the MIT shared memory extensions and by processes that want to concurrently access and change the same data in memory, usually databases.

Because processes are supposed to be able to directly access the shared memory without the intervention of the operating system, these shared memory mapping accesses are thus not under any direct control from the OS itself. Thus the operating system generally *cannot* enforce cache coherency on these kinds of shared mappings. The approach normally taken by Linux is to disable caching for such pages altogether, which incurs a very noticeable performance overhead on using shared mappings.

A possible approach to allow logical shared memory while avoiding disabling the cache is to use the page tables to serialize the access to the shared pages. The kernel

can protect any shared pages and allow only one process at a time to modify them. This way the kernel controls any shared resources and can make sure no aliasing ever happens by flushing the caches as appropriate when a new process needs to access the shared page. Thus the memory is not really ever physically shared, but the operating system makes it appear that way to user processes.

Using the virtual memory subsystem to serialize access potentially also allows distributed shared memory machines with this kind of software-imposed cache coherency mechanism (so-called *shared virtual memory*, [AMMR92]). However, the coherency overhead is huge compared to hardware coherency support, and performance is generally not acceptable for many traditional uses of shared memory (inter-thread lock handling, for example). In non-distributed systems it is generally better to disable all caching, as performance in that case will at least be predictable under different circumstances. For this reason no current Linux port uses page table shared memory serialization.

In general, virtual caches for data should be avoided as far as possible. The problems with shared memory areas make such caches inherently slower to handle, and while most applications do not need coherent shared memory, those that *do* need it get penalized. Note that because of cache aliases, virtual caches are also likely to incur more cache misses than physical caches — even without any coherency issues caused by writes.

It should be noted that most architectures that have virtual caches have only small virtual caches close to the CPU, where the latency of the address translation makes more of a difference. They then mostly have larger second- or third-level caches that are physically indexed.

What makes the approach of using virtual data caches even more suspect is that the address translation latency of physical caches can often be hidden with judicious cache design. One approach used by many physically indexed caches is to have a first-level cache that is indexed through the low-order bits of the virtual address that do not change during the virtual address translation — the page offset. This approach limits the size of the cache[11], but because the index does not change during the address translation, the cache lookup and address translation can be done in parallel [HP96, p. 425].

[11] Maximum cache size being limited to $pagesize_{CPU} \times associativity_{cache}$

4.4 Multiprocessor issues

While multiprocessing used to be the exclusive domain of supercomputing and large servers, advances in technology have meant that computers with more than one processor are becoming increasingly common. That has made these multiprocessor machines much more attractive to the Linux developers. At the same time, Linux itself has gained wider acceptance, which in turn makes Linux more attractive even to people with very high-end multiprocessor hardware.

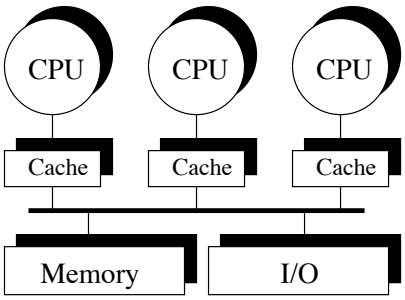

Figure 13: Centralized shared-memory multiprocessor

As a result, Linux has during the last two years been actively developed for these multiprocessor machines, and the issue of "portability" has not only implied changing the underlying CPU architecture, but the possibility to run Linux on different CPU configurations.

There are generally two separate types of modern multiprocessing environments, the decisive issue being how the main memory is accessed. The memory bus is generally the main limit in any multiprocessing environment (after all, it is often the limit even in uniprocessor environments), and the architecture of the memory subsystem is thus *the* major design issue.

The more common architecture is the centralized memory multiprocessor, where all processors share the same memory equally as in Figure 13. This is generally done with a common memory and I/O bus, and because all the processors have direct and equal access to all of the resources of the machine it is generally called SMP (*Symmetric Multi-Processing*).

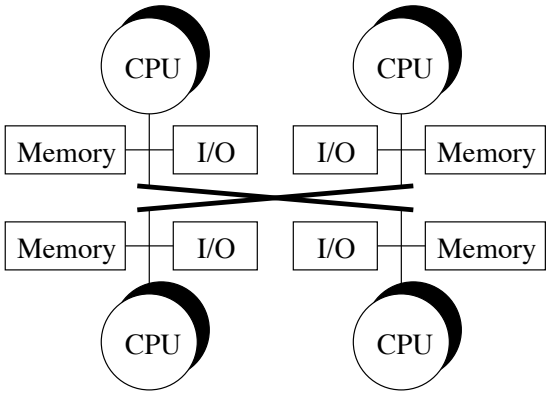

Figure 14: Distributed memory multiprocessor

While the shared bus approach is simple and thus cheap, it has the problem of not scaling very well to a large number of CPU's. With many CPU's on the same bus, the bus is quickly saturated and the approach generally does not scale to more than a dozen CPU's.

Because of the scaling concerns, the other main type of multiprocessor has a distributed main memory, with each CPU having a local memory (see Figure 14). The memory bus is no longer a common bottle-neck, and the CPU's are more independent of each other. As a result, performance of such a machine scales to much higher number of processors. The common term for these machines is MPP (*Massively Parallel Processing*).

It should be noted that physical distribution of memory does not necessarily imply that all the processors have separate logical address spaces. Many MPP-type machines provide for inter-processor communication through a globally addressable memory space. These kinds of distributed shared-memory machines are often called NUMAs for *Non-Uniform Memory Access*; all the memory is globally accessible, but depending on how far away the CPU is from the memory the access time differs. In that context SMP machines are generally called UMA (*Uniform Memory Access*).

Because the two approaches to multiprocessing result in fundamentally different behavior in memory accesses, the Linux kernel has two totally separate strategies for handling these machines.

4.4.1 Symmetric multi-processing

While porting an operating system from one CPU architecture to another at first glance appears totally different from the issue of running the operating system on multiple CPU's, the issues concerned are really basically the same. In both cases it is a matter of making the operating system run efficiently on a new hardware configuration.

To some degree even the problems are similar: the issues of initial bootstrapping of the system and the hardware details with regard to inter-processor signalling are in some respects similar to a port to another architecture. Similarly, the cache coherency issues in an SMP environment could be seen as just an extension of the cache coherency issues that can be seen with virtual caches and DMA on uniprocessor systems.

But even though there are similarities between porting an operating system to a new CPU architecture and a "port" from a single CPU to multiple CPU's, multiprocessing does add a few unique problems. The main issues are

- Data structure consistency; the need to protect internal kernel data structures from simultaneous accesses by multiple CPU's, generally through some synchronizing primitive.

- Allowing application programs to take full advantage of the presence of multiple processors.

In addition to the above, symmetric multiprocessing also exposes some other caches that are internal to the CPU, notably the virtual memory address translation cache (commonly known as a *Translation Lookaside Buffer* or *TLB*). Keeping the TLB synchronized with the current virtual memory mapping is trivial on a single-CPU system, because there is only one CPU that changes the translation, and it does so under very controlled circumstances.

With multiple CPU's, the virtual memory mapping changes have to be broadcast to any other CPU that may be using that mapping, and the CPU's have to invalidate the mapping before the original CPU can be sure that it can safely drop the old map.

Finally, with multiple CPU's sharing the same memory, the *ordering* of memory accesses is important. With caches between the CPU's and the system memory, the order in which any update from one CPU arrives at another CPU is by no means obvious. Especially in high-speed implementations where writes can be delayed on one CPU and the reading CPU may do aggressive pre-fetching of memory data are ordering issues very important.

Implementing an SMP kernel

The most obvious problem with any multiprocessing environment is the need to protect internal data structures from being changed by multiple different processors in an unpredictable manner. What makes the problem subtler is the fact that while unprotected data structures can be a major problem, the actual *symptoms* are generally rarely seen and hard to reproduce.

The basic problem is that updating non-trivial data structures cannot generally be done atomically, and some way of protecting other processors so that they do not see partially done updates is needed. While in some cases the problem can be minimized through using appropriate data structures where the final update can be done with one atomic update (so-called *self-locking* or *lock-free* data structures [PLJ94]), the generic answer is to protect the update through the use of locking.

However, using locks to avoid race conditions on data structures has its own set of problems — deadlocks and performance. Fine granularity locking can noticeably

impact performance if the operating system spends much time acquiring the necessary locks, and care must be taken that there is no possibility for circular lock dependencies resulting in deadlocks.

To make matters worse the issues of deadlocks and race conditions are often not well understood even by long-time programmers. Current programming languages and paradigms have resulted in most programmers having a very linear view of what happens in any given program, and concurrency is generally not well understood. As a result, any synchronization method must be simple enough to understand that subtle pitfalls do not make kernel programming too hard.

The current Linux kernel approach to kernel locks has been to adapt the simplistic approach of just one kernel lock. This approach is similar to the non-reentrancy rule of traditional uniprocessor UNIX — where a process running in kernel mode is never pre-empted by another process. The single-lock approach simply extends this requirement to multiple CPU's. Each time a kernel service is requested, the kernel lock must be acquired before entry into the kernel, and as a result internal data structure integrity is trivially ensured.

While the one lock approach is sufficient for ensuring data structure integrity and also does not lead to deadlock situations, it does have some obvious scalability problems. The approach is fine for a limited number of CPU's with computing loads that spend most of their time in user space with only very occasional excursions into kernel mode, but it scales very badly and has bad behavior for kernel-intensive jobs.

As a result, the current Linux kernel is not expected to be truly useful in SMP environments of more than 2-4 CPU's, and one of the current concerns is what the best balance of locking is in the kernel. Work is in progress to split the one lock first into a few smaller locks, and later pinpoint possible contention spots in those.

It is hoped that reasonable scalability can be achieved without excessive numbers of locks, especially as the current goal for good scalability is for reasonably few processors. While bus-based hardware can scale to more processors, cost-efficiency seems to be much higher for few processors and four processors seems currently to be a good goal. For really high-end hardware the solution is generally not SMP in any case (see chapter 4.4.2).

4.4.2 Massively parallel systems

While small-scale SMP shared uniform memory machines are becoming common, they do not offer the scalability required for really high performance computing. With tens or hundreds of processors, massively parallel processing systems fill that niche, and Linux is increasingly attractive for these kinds of systems too.

However, the same hardware scaling considerations that make a uniform global memory access machine impractical for many processors also make the software locking approach to SMP impractical. While software could in theory scale better, practical concerns about deadlocks and lock granularity (see the previous chapter) make it questionable whether such an approach is viable.

The software solution to the scaling problem is exactly the same as the hardware solution was: de-coupling the nodes from each other. Instead of running the same operating system image on each and every node, each node gets a local copy of the operating system the same way each node has its own memory store. The different operating system kernels communicate with each other over the high-performance internal network, but can at the same time be seen as independent entities.

Note that SMP and MPP are in no way mutually exclusive approaches, and indeed one of the most interesting approaches is to combine the two into a hybrid system. Such a hybrid system would consist of multiple independent nodes connected together with a high-performance network, where each node would contain a small number of processors in an SMP configuration.

While such a hybrid system running Linux does not yet exist, there actually are two separate high-performance MPP projects using Linux as the base operating system. The first project was *Beowulf* [BSS+95].

The Beowulf project at NASA CESDIS (*Center of Excellence in Space Data & Information Sciences*) research center aims to create cheap high-performance computing hardware through the use of common off-the-shelf hardware, and using Linux as the operating system. The nodes are built up from normal PC hardware, connected together with 10 and 100Mbps ethernets.

The second project is being developed by the CAP research project at the Australian National University in Canberra [TMSW96]. In contrast to the Beowulf project, this AP1000+ system is a decidedly non-off-the-shelf MPP made by Fujitsu, using 50MHz SuperSparc CPU's. Rather than using off-the-shelf networking, the AP1000+ has a powerful internal network with very low latencies and high

bandwidth. While the maximum configuration is able to incorporate 1024 of these CPU's in one machine, Linux has so far been run on only 16, 32 and 64-cell machines.

Despite the very different hardware base and the differing aims of the projects, the basic approach has been similar in both projects. While some support for the MPP hardware has been added to the basic kernel, most of the distribution is done in user space through standard parallel programming interfaces (*Parallel Virtual Machine* — PVM and *Message Passing Interface* — MPI). This way most of the real complexity of a distributed environment can be handled at an appropriate level, leaving the kernel itself reasonably simple.

Note that this approach is fundamentally different from distributed systems based on microkernels. Rather than handling the complexity of distribution at a low level in the kernel, the distribution is done almost entirely in user space with only minimal kernel support. This allows for much greater flexibility, as the different nodes need not necessarily run the same operating system or even have similar hardware.

4.5 Device drivers: accessing the I/O bus

We have so far only concentrated our attention on the CPU subsystem of the portability issue. To some degree the CPU subsystem is the most visible part of any architecture, and the I/O characteristics of the hardware are often overlooked. Thus, for example, we talk about Linux/Alpha and Linux/x86 — corresponding to the CPU architectures Linux has been ported to — but never about Linux/sbus and Linux/PCI and so on.

One reason that the I/O architecture is often overlooked is that it should never be seen directly by any application programs. While people are used to having to have separate versions of their binaries for different CPU architectures, the same is not true when it comes to I/O. The I/O details are expected to be totally hidden by the operating system, and while differences in the I/O architecture is expected to make a difference for performance, it is not expected to show itself any other way.

Yet, despite this lack of user attention, the I/O details can easily be the largest part of any portability effort. While compilers and other virtual machine abstractions can hide the CPU details quite effectively, there are generally no "I/O-compilers" available to hide the details of various devices. As a result, much of the device driver portability work has to be done by hand, from scratch.

What often makes the issue even thornier is the lack of documentation on the details of the I/O architectures. While all the relevant details of a new CPU architecture are generally readily available in just one or two architecture manuals from the manufacturer of the CPU, the same is seldom true of the I/O details. Often the person doing the port is forced to gather the information from many separate (and often incomplete) sources.

4.5.1 Proprietary buses

Making the issue of I/O architecture portability harder is not so much the general complexity of the architecture, as the sheer overwhelming volume of details pertaining to the I/O subsystem. Only a few years ago, every major manufacturer had their own I/O bus standards (and many had several different standards), the same way they had their own CPU architecture.

But while the CPU is one well-defined entity, the I/O architecture is not only defined by the details of actually driving the physical bus, it is also defined by the devices connected to the bus. Often potentially *thousands* of different devices with no common interface should be supported. And unlike the CPU, most of those devices tend to have failure modes that have to be handled and recovered from.

In fact, so overwhelming is the amount of device driver code that almost exactly half of the current Linux kernel sources is composed of just device drivers. While few of the drivers on their own are overly complex they are numerous, and a large portion of the porting effort has been involved with device driver work.

Making all the problems worse is the fact that common code shared between drivers is usually very scant. There exists various common driver layers, but much of the common driver layer is not so much concerned with the driver itself as with the common interface all drivers have to show to the rest of the world.

One example of a generic driver layer is the UNIX `tty` layer, the layer concerned with serial input devices like a keyboard or a character terminal. This layer contains generic code that is required for any `tty` device — character translation, session handling and actions to be taken on exceptional events like a hang-up of the line. However, the entire common code is concerned with issues that are needed for generic UNIX compatibility, not so much the low-level details of the actual device.

Happily, some real standards do exist in the jungle of I/O handling. The main such standard is the SCSI (*Small Computer Systems Interface*), which offers a common hardware bus standard used by various devices ranging from magnetic and

optical disks to scanners and tape drives. While the SCSI standard does not specify the actual programming interface as seen by the operating system, the common standard allows for common code to be shared across different SCSI adapter drivers.

4.5.2 PCI — the emerging standard

While the situation with respect to I/O architectures is very complex, and very little common code can generally be shared between different I/O subsystems, the last few years have introduced a new standard that may put an end to at least some of the chaos. That new standard is the PCI (*Peripheral Component Interconnect*) bus standard introduced by Intel and other computer companies.

What is interesting about PCI is not the technology itself, but rather the potential for a reasonably common standard. Already, PCI has been revolutionary in that it has been able to attract many different vendors to use the same standard rather than continue to use proprietary standards not shared with anybody else. Though the PCI standard initially was embraced mainly by PC manufacturers, the large body of available PCI-compliant hardware that it resulted in has made PCI an extremely attractive proposition to other vendors too.

Currently, PCI is used both by ix86-, PowerPC- and Alpha-based systems, and lately Sun has announced that PCI-based Sparc workstations are going to be available in early 1997. What is so impressive about this list is that these four major architectures not only happen to be the primary Linux platforms, they also account for a large portion of the modern CPU market. As such, PCI may reduce the amount of I/O architecture portability work very noticeably.

It should be noted that while a move to PCI makes the portability issues of the actual I/O bus interface easier, it is by no means the answer to all problems. Most notably the very large PC hardware market not only made PCI a viable standard in the first place, it has also resulted in unprecedented numbers of different adapter cards being available for PCI. As such, while there is a common bus that avoids one level of portability problems, it certainly has not taken away the wide variety of adapters, and thus the need for device driver development in the future.

5 Example Architectures

While Linux currently supports seven architectures to some degree or other (Digital Alpha, Intel 80x86, Motorola 68k, MIPS, PowerPC, Sparc, Acorn ARM), three of those architectures stand out by having had a fundamental impact on Linux portability.

The original Intel platform was obviously the base of any Linux work: it offers cheap and ubiquitous hardware and is still by far the most popular Linux platform and the one most development is done on.

At the same time, the Digital Alpha platform stands out as a modern 64-bit hardware base, and being the first target of a Linux port. Together with the Sparc architecture, these two architectures have defined most of the Linux portability work so far.

5.1 The Alpha

The Digital Alpha architecture is one of the most recent RISC designs in the industry, and, thanks to extensive support from Digital Corporation, was the first non-Intel architecture supported portably by Linux.

What makes the Alpha such an intriguing platform for Linux is not only the high performance of Alpha-based computers, but also the fact that an Alpha port was the ideal second platform from a portability standpoint. Not only is the Alpha architecture fully 64-bit both from a arithmetic and a memory management standpoint, it is a very aggressive RISC architecture that looks very different from the original platform for Linux, the Intel x86 [Dig94].

What made the Alpha an even better porting target was the fact that while having a markedly different CPU architecture, Digital was one of the first to embrace PCI as the standard for their I/O bus hardware even for non-PC markets. As a result the initial port did not have to overly concern itself with device driver issues, allowing the project to concentrate on the design of the CPU architecture itself.

The Linux/Alpha porting project actually started out as two separate and independent projects: one pilot project inside Digital itself to prove that Linux could be run on Alpha hardware, while at the same time the author was given access to Digital Alpha machines for another port.

While that early porting effort inside Digital was concerned with making Linux run on the Alpha (somewhat akin to the original Motorola 68k Linux project), work

at the Department of Computer Science of the University of Helsinki concentrated on creating the basis of a truly portable operating system. As such, the Digital project was working on a 32-bit port on the assumption that it would be easier, while the author worked on making Linux port cleanly to a full 64-bit architecture and take full advantage of the hardware provided.

As it turned out, the 32-bit Linux/Alpha port was bootable at an earlier date, but the portable 64-bit clean rewrite was not much behind and quickly proved itself to be more versatile. While the Digital Linux efforts have not stopped, they have abandoned the early 32-bit pilot project and are now working on making Linux available on different Alpha hardware and providing the required environment for the kernel. An SMP effort is also underway to take advantage of multiprocessing capabilities of the Alpha.

The main changes introduced by the Alpha port to Linux have been

- Re-organization of the kernel into architecture-independent and architecture-specific parts. While the original kernel had used inline assembly statements embedded in the sources, portability required a clean and well-defined interface for code that needed to access specific hardware features.

- Virtual memory management re-write. The wider address space required a deeper kernel page table tree, and together with generic portability concerns resulted in the current memory management implementation.

- Kernel type cleanliness. The wider data types required major cleaning up of the kernel internal data structures. While the original Linux kernel had worked with the C types "`int`" and "`long`" interchangeably and used both of them for 32-bit values, the Alpha required a clean separation of the different data types used in the kernel.

- Strict alignment of kernel data structures.

In addition, the Alpha port also exposed some unportable assumptions in the kernel itself. For example, the Linux kernel originally assumed that byte accesses were atomic, and in some cases used byte values for flagging asynchronous events. On the Alpha, the smallest atomic update is to a 32-bit word, and all byte accesses are done with multiple instructions. As a result, code that depended on atomic updates broke subtly.

There were other subtle issues involved with the Linux port to the Alpha. For example, the absence of byte accesses is not visible to the C programming level

as the compiler will automatically generate the correct instruction sequences, but for performance reasons contiguous byte accesses should be shunned in favor of optimized 64-bit code. Rewriting some basic library functionality as optimized assembly made a noticeable difference in speed for some uses.

The Linux/Alpha port also introduced the issue of native binary compatibility with another operating system. Being binary compatible not only gave Linux/Alpha access to Digital UNIX binaries, it also helped speed up the system bootstrap process by being able to test out early versions of the operating system against binaries that were known to work.

The issues of different data sizes on the Alpha due to the 64-bit architecture, together with the binary compatibility requirements, made it important that all the kernel interfaces were able to handle different type sizes transparently. One obvious example of this was the kernel access to user mode data.

Before the Linux/Alpha port, writing a data item to user mode was done with the functions `put_fs_byte()`, `put_fs_word()` and `put_fs_long()` — for 8, 16 and 32-bit entities respectively. This mapped well to the original platform data types. The strange naming of these functions came from the use of the `fs` segment register on the Intel platform to access user mode memory.

On the Alpha, this obviously did not work very well. Not only was the naming non-intuitive on the Alpha as there is no `fs` segment register, but a data type that was 16 or 32-bit on the Intel platform might be 32 or 64-bit on an Alpha. As such, the whole concept of fixed size data had to be scrapped, and the interface to access user mode was renamed as just `put_user()`. This macro automatically determines the right data size at compile time and uses the appropriate method for the access. These kinds of changes made it possible to use the appropriate types on different systems without having to worry about how the actual access happens.

Linux/Alpha, being one of the older ports of Linux, is a very stable platform, and is used widely in various environments that need superior floating point performance or need access to 64-bit hardware. The port has been entirely merged with the standard Linux sources.

5.2 The Sparc line of computers

While the Alpha architecture lay the groundwork for Linux kernel portability, what cemented it down was the port to the Sparc line of computers led by David Miller.

Two things make the Sparc architecture so interesting in this context. First, the hardware is readily available: especially older Sparc hardware can generally be found in various dark corners of any university. In that sense the availability of hardware was to some degree similar to the availability of standard PC hardware.

The other thing that makes the Sparc an interesting platform is that the Sparc architecture is one of the oldest RISC architectures, and has gone through several generations at the hands of several different manufacturers. And while the different versions of the architecture look similar to user mode programs, the differences on a system programming level are startling. As such the Sparc port in one sense contained several "micro-ports", and brought up many issues that had not been problematic on the previous two architectures [SPA94a, SPA94b, Ros93].

The Sparc, together with the Motorola 680x0 port, also introduced a different byte order for Linux. While both the Intel 80x86 and the Digital Alpha architectures are little-endian, the Sparc and the m68k are big-endian. Also, not only have different versions of Sparc CPU's used various different virtual memory management techniques, they have also used both virtual and physical caches or mixtures of them both.

While this wild variety makes porting difficult, it also makes the Sparc an excellent testing environment for the kernel virtual machine. The fact that the Linux/Sparc port not only worked, but actually was very successful indeed is a good indication that the Linux virtual machine is generic enough to handle any reasonable hardware implementation.

The biggest changes introduced by the Sparc port to Linux were in memory management and byte order. The byte order changes were mostly related to file system code — notably the native `ext2` file system and the MS-DOS file system. The memory management changes were mainly caused by performance and cache invalidation issues. Both the Intel 80x86 platform and the Alpha handle data cache coherency issues in hardware, making the Sparc port the first to tackle this issue in software.

Also, while the virtual machine three-level page tables turned out to be a very good design, performance issues required that the page table TLB invalidation be made much more granular for the Sparc. The original Linux invalidated the whole TLB whenever any invalidation was needed, and that was later extended to invalidate single page translations in some circumstances. However, on the Sparc this two-level granularity was inadequate.

The reason for needing finer granularity invalidates on the Sparc is the size and complexity of the TLB. On the Intel 80x86 and the Alpha, the TLB consists of only the on-chip TLB that can easily and quickly be invalidated and quickly re-filled. As such, while some care is needed, TLB invalidation generally is not much of a performance issue on these architectures. In contrast, on some Sparc hardware the CPU uses a hash table in memory to expand the on-chip TLB, and invalidating this is a much more costly operation.

As a result, an in-memory TLB extension as on the Sparc and the PowerPC requires much finer granularity control over TLB invalidates. The in-memory TLB is much larger — thousands or even millions of entries as compared to a on-chip TLB of hundreds of entries at the most.

Similar granularity concerns faced the cache coherency code, resulting in the following interfaces for flushing caches and TLB entries:

`flush_cache_all()`, `flush_tlb_all()`: flush all entries under exceptional circumstances (at boot-up and when the kernel mapping itself changes).

`flush_cache_mm()`, `flush_tlb_mm()`: flush a whole process cache and TLB when the process undergoes radical change like executing a new binary image.

`flush_cache_range()`, `flush_tlb_range()`: flush a range of pages in a process, for use when a part of the process map changes due to a new mapping.

`flush_cache_page()`, `flush_tlb_page()`: flush a single page.

On hardware where cache coherency is handled by hardware, the cache flush operations will be no-ops and will be removed by the compiler. Similarly, on most CPU's with just an on-chip TLB, the TLB range invalidate generally will invalidate the whole process TLB — the more complex partial invalidation simply is not needed (nor supported by the hardware).

The Linux/Sparc is now, like the Alpha port, one of the most stable platforms for Linux, and is used in various real world situations[12].

[12] As of November 1996 most of the Linux specific mailing list traffic has been handled by a Sparc sun4m machine running Linux at `vger.rutgers.edu`.

6 Future Work

While the portability phase of the Linux development slowly has been calming down, and most of the issues have been resolved, not all of them are by any means finalized.

The basic Linux kernel has been clearly shown to be very portable, as proven by multiple ports to varied platforms. The different platforms supported have different CPU's, memory interfaces and IO buses, and often have their own software binary standard requirements as well. All of these issues have generally been resolved already, and in that sense the porting effort has clearly been a complete success.

However, there still exist implementation issues within the Linux kernel that directly impact portability concerns. These are often cross-cutting issues where performance requirements in other areas require new hardware interfaces and thus extensions to the virtual machine.

One issue in particular is the use of the virtual memory management unit to do data copying in high-bandwidth applications. With memory speeds often being the bottle-neck for high performance computing, the use of virtual memory re-mapping can be a way to avoid physically copying data.

While the current Linux memory management has proved to be an efficient and portable design, optimization issues like the above lead to new concerns. When is such an optimization worthwhile, and when should it be discarded in favor of a traditional physical memory copy? These issues depend not only on how much of a bottle-neck the memory copy itself is, but also on how expensive the virtual mapping change can be. That in turn depends on the virtual memory hardware, but also on the design of the cache subsystem where a virtual cache can result in new cache coherency issues that have so far not been a concern.

Another area that needs work is the hardware abstraction of multiprocessing in the Linux kernel. The current SMP effort only supports the Intel 80x86 and the Sparc platforms, and further work in this area is required. What is the impact of hardware with no memory coherency on SMP? What is the proper locking granularity, and what is the impact of hardware with a larger overhead for SMP-safe locks?

Also, while the basic design has been proven to be very portable, we expect to spend much of next year on low-level expansion of the ports, and getting the different architectures up to the same level of support that is currently offered on i386, Alpha and Sparc platforms. The portability work is by no means done.

Finally, it should be noted that the portability work has in no way affected the original Linux/Intel version adversely. Not only is the current portable Linux totally compatible with the original Linux, but the porting effort has also forced cleaning up of the kernel code. As such the result is a cleaner kernel with much better abstraction of the hardware side.

Also, good design of the virtual machine has not only avoided a negative performance impact — performance has actually improved. This is mainly due to other performance-related efforts done in parallel with the portability work, but the better abstractions required by portability issues have resulted in better interfaces that have been easier to optimize.

The original three design goals — simplicity, efficiency, and compatibility — have been joined by a fourth one, portability.

References

[AMMR92] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. In *ACM Operating Systems Review*, volume 26, pages 4–26, July 1992.

[App96] Apple Computer, Inc. *MkLinux: Linux for Power Macintosh*, 1996. `http://www.mklinux.apple.com/`, checked January 24, 1997.

[Bäc96] Ralf Bächle. The Linux/MIPS FAQ, 1996. `http://www.unix-ag.uni-siegen.de/~nils/mips/linux-mips.html`, checked January 26, 1997.

[BGG+91] Allan Bricker, Michel Gien, Marc Guillemont, Jim Lipkis, Douglas Orr, and Marc Rozier. A new look at microkernel-based UNIX operating systems: Lessons in performance and compatibility. In *Proceedings of the EurOpen Spring 1991 Conference*, Norway, May 1991. `ftp://ftp.chorus.fr/pub/reports/CS-TR-91-7.ps.Z`, checked December 31, 1996.

[BSS+95] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: a parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, USA, 1995. `http://cesdis1.gsfc.nasa.gov/linux-web/papers/ICPP95/final.ps`, checked December 31, 1996.

[CG87] John H. Crawford and Patrick P. Gelsinger. *Programming the 80386*. SYBEX, USA, 1987.

[Dig92] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha Architecture Reference Manual, Part III: DEC OSF/1 Alpha Software*, 1992.

[Dig94] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha AXP Architecture Handbook*, 1994.

[Dou96] Cort Dougan. Linux for PowerPC, 1996. `http://www.cs.nmt.edu/~cort/linuxppc/`, checked January 26, 1997.

[FL94] Bryan Ford and Jay Lereau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Conference*, pages 97–114, USA, January 1994. `http://www.usenix.org/publications/library/proceedings/sf94/ford.html`, checked December 31, 1996.

[GC94] Berny Goodheart and James Cox. *The Magic Garden Explained: the Internals of UNIX System V Release 4*. Prentice Hall, Australia, 1994.

[GM95] James Gosling and Henry McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, Mountain View, CA, USA, October 1995. `ftp://ftp.javasoft.com/docs/whitepaper.A4.ps.tar.Z`, checked January 24, 1997.

[Hoh96] Michael Hohmuth. Linux-Emulation auf einem Mikrokern, 1996. `http://www.inf.tu-dresden.de/~mh1/prj/linux-on-l4/`, checked January 24, 1997.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[IKWS92] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The effects of virtually addressed caches on virtual memory design and performance. In *ACM Operating Systems Review*, volume 26, pages 14–29, October 1992.

[Ins96] The Institute of Electrical and Electronic Engineers, Inc., USA. *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1996.

[Int91] Intel Corporation, USA. *Intel386 Family Binary Compatibility Specification 2*, 1991.

[Int93a] Integrated Device Technology, Inc., Santa Clara, California. *IDT79R4600 ORION Processor Hardware User's Manual version 1.0*, October 1993.

[Int93b] International Business Machines Corporation, USA. *PowerPC Architecture*, first edition, May 1993.

[Kin96] Russell King. ARM Linux, 1996. `http://whirligig.ecs.soton.ac.uk/~rmk92/armlinux.html`, checked January 26, 1997.

[Law97] Chris Lawrence. The Linux/m68k Home Pages, 1997. `http://www.clark.net/pub/lawrencc/linux/index.html`, checked January 24, 1997.

[LHFL93] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proceedings of the Third USENIX Mach Symposium*, pages 39–55, USA, April 1993. `http://www.usenix.org/publications/library/proceedings/mach3/lepreau.html`, checked December 31, 1996.

[PLJ94] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *IEEE Transactions on Computers*, volume 43, pages 548–559, May 1994.

[Ros93] Ross Technology, Inc., Austin, Texas. *SPARC RISC User's Guide HyperSparc Edition*, 1993.

[SPA94a] SPARC Technology Business, USA. *microSPARC User's Manual*, 1994.

[SPA94b] SPARC Technology Business, USA. *SuperSPARC and MultiCache Controller User's Manual*, 1994.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1992.

[TMSW96] Andrew Tridgell, Paul Mackerras, David Sitsky, and David Walsh. AP/Linux — a modern OS for the AP1000+. In *6^{th} Parallel Computing Workshop*, pages p2–c–1 – p2–c–9, Japan, November 1996. `http://cap.anu.edu.au/cap/projects/linux/aplinux-pcw96.tex`, checked December 31, 1996.

[Tor93] Linus Torvalds. Linux—a case study in free software development. In *Proceedings of the NLUUG autumn conference 1993*, pages 133–140, Ede, The Netherlands, November 1993.

[WHL93] C. Eric Wu, Yarsun Hsu, and Yew-Huey Liu. A quantitative evaluation of cache types for high-performance computer systems. In *IEEE Transactions on Computers*, volume 42, pages 1154–1162, November 1993.