# Chapter 1. Linux is not Unix (20)

GNU is not Unix and Linux is not Unix. We might as well get that straight from the start. Whatever was true about the old Unix operating system does not necessarily apply in this book and taking any such assumption probably does more harm than good. In fact, Jon `Maddog` Hall tells us these days that it isn't even legal to say that Linux is "a Unix-like operating system" because we have never subjected Linux to the standards tests or applied for any such designation. Maddog suggests we say instead that "Unix is a Linux-like operating system".

Unix and Linux share architecture to a point, but they are very different creatures, fundamentally different in their design and development. Unix was a serious effort, the outgrowth from only a few prior prototypes, intended from the start to be a commercial grade operating system; Linux began as a toy of interest to only a handful of friends and has grown to the present 1.6 million lines of code entirely incrementally, and almost exclusively in the *maintenance phase* of its development. Unix was largely carved in stone before anyone had the chance to deploy it in the real world, whereas Linux is the sum of countless solutions and revisions, each done in direct response to practical, real-world problems. That Unix resembles Linux in a superficial sense of multitasking and multiuser operations is at best proof of Jack Scarfatti's thesis on how the future causes the past.

## A Brief History of Linux

Like any technology born and evolved on the Internet, the folklore of Linux is very well documented; only the exact date of the first compile of the first toy system which Linus released to a few friends in October of 1991 has been lost. What is important, though, is to realize this was a toy, a proof only of a multitasking environment for the Intel 386 platform, and one man's response to the mess which had resulted from the Proprietary Era of software development.

### From Multics to the Unix Wars

Thirty years before Linus released Linux 0.02, the Compatible Timesharing System (CTSS) for the IBM 709 emerged from MIT as the first of the time sharing systems; CTSS ran multiple users and swapped to magnetic tape and grew to allow 30 concurrent users. In 1963, under Project MAC and funded by ARPA and the US Department of Defense, the Multics project was established to investigate multiple access computing. Early partners in Multics included Bell Labs and GE/Honeywell and by 1965, Bell Labs were assigned to producing the PL/1 compiler for the Multics system.

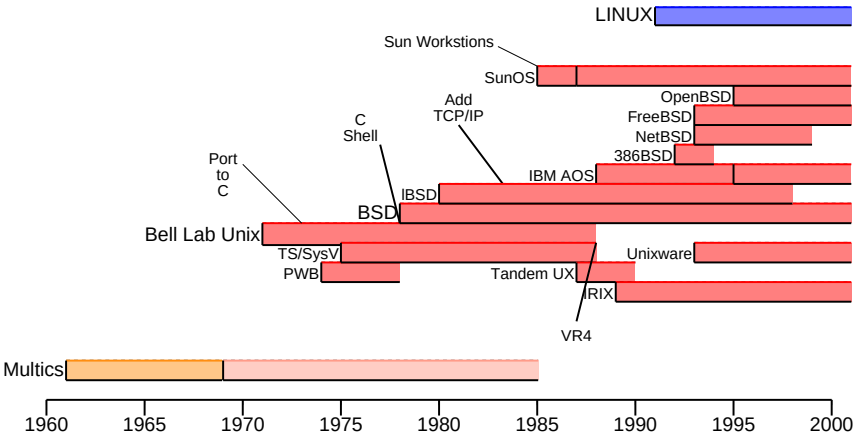
Bell Labs left the project in 1969. Although Multics continued to explore ideas in virtual memory, multiple disk volumes and relational databases and lasted well into the 1970's, there was a reluctance to commit to an experimental system; after a brief renaissance in the late 1970's, Multics applications were gradually migrated to Sequent, and by 1995 only one Multics system remained. Bell Labs were

convinced Multics would fail to deliver, and the last participants, including Dennis Ritchie, left to explore ideas in communal computing aimed at producing remote-access, time-sharing machines which would encourage communications.

During 1969, Thompson, Canaday and Ritchie laid the design work and built simulations for what would become the Unix inode file system, device files, memory paging and the command shell. Their system was implemented on the PDP-7 and, by late 1970, Brian Kernighan dubbed the emerging system "Unics". In 1971, migrated to the PDP-11 and sporting modern process control, a port of the Multics roff and a simple editor, Unix was delivered to the Bell Labs Patent Office. By 1973, the kernel was re-implemented using the new, higher-level and more portable C programming language and was extended to multi-programming; V4, the first modern Unix was born.

V6 was released outside of Bell in 1975 and became the basis for the first commercial Unix implementations and the first 1.xBSD from the University of California at Berkeley. BSD added the C shell followed a year later by the AT&T Bourne Shell in V7. Sun Microsystem came online in 1983 with the first Unix workstations, and 4.2BSD in 1984 added complete TCP/IP networking protocols; Unix now integrated multi-user multi-tasking with pipes, shells, IPC, remote logins, uucp, file transfer and electronic mail.

**Figure 1-1. Unix Time-line 1961-2001**




Major developments in multi-tasking computing from Multics through to Linux.

By 1991, Unix was in trouble. In bids to "conquer" market segments or mind share, or just on matters of principles and licenses, the original Unix forked and fractured into three camps, roughly the AT&T (owned by Novell through USL), OSF and BSD, and each of these into a dozen small factions; BSD became BSD, OpenBSD, NetBSD and FreeBSD, V7 because System V, Tandem UX and VR4; hybrids beget IBM AOS and Unixware and the ironically named Consensus. Then came IRIX, Solaris, HP/UX, AIX, QNX, XENIX and great hoards of others. When asked about the danger of a similar fork of

diverging Linux kernels, almost all developers will point to the “Unix Wars” as the best reason why no one would *want* to inflict the same suffering on Linux.[1]

## From Task Scheduling to Open Source

Such was the mess which greeted Linus Torvalds in his search for a platform to do multi-tasking research, and this was the world awaiting the multi-tasking kernel he would inadvertently supply. Unix was rapt with license wars and infighting, Andy Tannenbaum’s Minix was a case of “Close but not quite” and was under a restrictive license. Putting this all together, Linus wrote his project from scratch, placing the result under the GNU Public License, and emailed his work to a handful of friends on October 12, 1991.

# The Holy Grail of Portability

While there was no shortage of contrary opinion, one reason Ken Thompson and Dennis Ritchie chose to implement Unix in a high-level language to achieve a level of abstraction from the underlying hardware. The choice was done partly for the convenience of the C language, but also for the promise of an easier port to other platforms. It proved a useful experiment: By the early 80’s, the only component of Unix still coded in assembler was the assembler itself. The ease and portability of C was in large measure responsible for the rapid spread of Unix.

Unix design philosophy also called for abstraction in hardware devices, virtual memory and in file systems, always in a trade-off between what can be wrapped in an abstraction layer and performance requirements.

This philosophy persists in Linux where a prognosis for any port is positive where there exists a working GNU C compiler.

## Designing to a “Reasonable Architecture”

High-level languages and open standards help, but they no guarantee of portability. In the earliest port of Linux, to the Motorola 68k CPU, many changes were made and whole sections re-written to accommodate future ports. In his essay in “Open Sources, Open Minds”LT99, Linus Torvalds describes the design objective of Linux as seeking not to be portable, but to be portable to “any reasonable architecture”.

Rather than seek universal portability, as was the Holy Grail of the Microkernel school of O/S design, Linus designed for a more restricted target architecture with the assumption that any *reasonable* hardware design must include basic features such as interrupts, memory and task management.

## Monoliths and Microkernels

Linux is a monolithic kernel, meaning the kernel draws a hard boundary between what is kernel space, accessed by direct function calls, and what is user space, accessed through abstraction layers such as system requests. In a microkernel, all system calls are messages and the microkernel architecture depends on automated optimization of these calls; in a monolithic kernel, this optimization is done by the programmers; where speed or timing is critical, methods are implemented as direct C function calls inside “kernel space” and using the memory and other resources of the core kernel.

In principle, this hard boundary poses a problem for the monolithic kernel: Extending the set of services requires extending the kernel itself. In Linux, this problem is alleviated through the technique of modules, of wrapping a thin abstraction layer around kernel services which allow the components to be loaded and unloaded at runtime, and which restrict the communications of the modules to the resources inside kernel space through a small set of strictly defined APIs. Strictly speaking, Linux is probably not a monolithic kernel but somewhat of a hybrid or middle ground design where trusted modules may exist on the gray boundary between user space and kernel space.

By definition, the modules are *trusted components* of the kernel, able to directly access hardware resources, to manipulate the state of the kernel, and, if they are not careful, to do a lot of damage. New in the 2.4 Linux kernel is a notion of access control for kernel modules, a security layer to assure the system operators that no unauthorized kernel services may be added to the running system.

# Open Sources, Open Minds

There are many reasons for the success of Linux; technical excellence is only one dimension and is itself a product of the decision to release first under the GNU Public License and second to open the development team to any and all contributors. This second choice set Linux apart from the prior GPL work done by the Free Software Foundation and is the reason why this book is about Linux and not about the Hurd. Whether through his insightful understanding, or through blind good luck, Linus Torvalds could harness massively parallel independent exploration and development of kernel components; 8 years later, this method continues to search the solution space unabated, percolating the findings of countless contributors up through the hierarchy of the `MAINTAINERS` list until the cream of the ideas crop show up in Alan’s `-ac` patches, and from there most often into the canonical sources.

## Linux Development Model

While he probably thought nothing of it at the time, Linus set this new project apart from the fold in his choice of an open and egalitarian development model. While Linus remains the final veto power over any new features vying for the canonical source tree, there are no restrictions on contributors; where prior “open source” projects would welcome advice (and patches), each restricted all write-access to the source

code to a tightly cloistered (and often geographically proximate) and democratic work group. Linux, on the other hand, followed the model of a “benevolent dictator” who would freely grant write access to anyone who proved themselves worth. Thus, while the FSF teams tend to cluster around MIT and Apache clustered around California, and while the developments in those projects were relatively slow, Linux became a true participatory open source project propelled by an open welcome for participants.

## Linux and the GPL

The GNU Public License did not begin with Linux, nor is Linux even its most famous product. The GPL was a stroke of Genius by Richard Stallman, author of the Emacs editor and the founder and primary contributor to the Free Software Foundation. Under the GPL, Emacs had become standard equipment for software developers and document management systems under Unix, and by 1988, the GPL GNU C compiler had become so entrenched in research and corporate computing that the FSF was pressured to release a modified license to allow for the development of proprietary software using these tools.

The GPL is available with the Linux source code and should be read thoroughly before embarking on any Linux-related project. The simple story of it is that it grants users of the software the rights to read the source code of the program (which is the extent of the definition of “open source”) and that it also grants all users of the software the right to modify and to *redistribute* this software without royalty fees. Most importantly, the GPL ensures that all modified versions of the GPL software are also themselves governed by the GPL.

## Lines in the Sands over GPL

The GPL is described as having “virus-like” qualities in that if any component of a software project contains GPL code, the whole body of work must also conform to the GPL. In the middle 1980’s, concerns over this effect led Richard Stallman to concede a “Lesser GPL” which prevented library functions linked during a C compile from determining the license of the resulting code; the LGPL thus opened the use of the GNU C compiler for commercial products and resulted in an explosion of popularity for the FSF compiler.

The development of Linux came to a similar line in the sand during the development of the 2.0 kernels: Extensions to the kernel, by being extensions rather than “clients” of GPL software, would be required to conform to the GPL. By public decree, Linus Torvalds announced that *modules* should be considered as “clients of the kernel” and not as components of the whole. Although there are kernel services implemented as modules which are included with the canonical kernel sources and to be included in the sources, a module must, by definition, make the source available (and under the GPL), there is no restriction against creating commercial and proprietary binary-only modules.

# Unix Foundations

Linux is not Unix, but borrows heavily from the design principles of Unix and emulates Unix interfaces, most often more accurately to spec than any commercial true-Unix. The early development of Linux was obviously influenced by the widespread use of Unix by its developers, but also because of the work done by the Free Software Foundation in creating a reasonably complete GPL opus of standard Unix tools; once the Linux kernel was only just useful enough to emulate the actions of the 1971-vintage Unix, the developers quickly realized they could use the ported GNU C compiler to build the Free Software Foundation GNU utilities and thereby assemble a surprisingly complete operating system.

Bundling GNU with Linux likely had a major influence in the emergence of the kernel interfaces; being a volunteer project, there was no incentive to re-invent methods, and Unix interfaces were well understood, well documented and came with a large body of applications. Thus Linux came to share the design requirements of Unix systems, and, unencumbered by profit motives, could do so in an open and accommodating way. Most of the programmers themselves were experienced in programming for the 4.3 BSD Unix kernel, and even very early kernel sources show many sections implemented or patched to match the quirks in 4.3 BSD.

The elements of the architecture grew naturally from this initial coupling with the FSF tools and the BSD environment. To be interface compatible, the new Linux would need

- Hardware abstraction through a small number of interfaces (`/dev`, `/proc` and `ioctrl()`)
- An i-node based file system
- Multi-tasking process control
- Virtual memory
- Kernel time keeping
- An i-node based file system

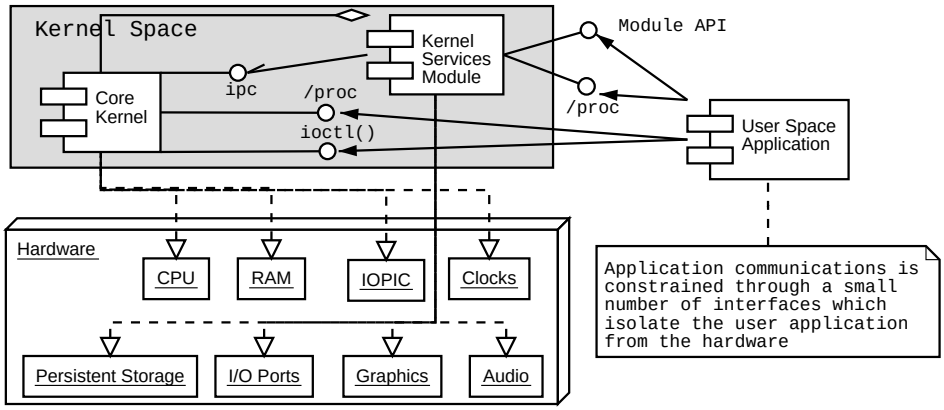
Given these requirements, and the constraints of compatibility with the FSF and BSD interfaces, Linux had one further constraint: It had to run on the Intel 386 microprocessor. Many attempts had been made prior to Linux to implement a multi-tasking, multi-user O/S on this platform; there had been some minor successes with DesqView and XENIX, but the sword remained firmly stuck in the stone until Linus Torvalds who made the innocent and reasoned choice to eschew popular opinion and implement his O/S as an old-fashioned monolithic kernel.

The sections which follow outline these interface and performance requirements in more detail, and each is also covered more exactly in the associated chapter.

## Hardware Abstraction

To be portable, software must divide the requirements of a system from the implementation details; in Linux, all hardware devices, from virtual memory to USB ports, are idealized in limited interface definitions. Of all aspects of kernel customizations, these API standards are probably the hardest to change and the development community is loath to allow feature-creep and bloat in the core kernel.

**Figure 1-2. Hardware abstraction**




Hardware abstraction is achieved through a limited number of interfaces from User Space to the kernel and from the kernel to the underlying hardware. While a module is free to define its own user space interfaces, developers are encourage to use standardized methods to ensure system security and portability.

## File systems

File systems in Linux are grounded in the i-node model first developed for the Thompson and Ritchie pre-Unix PDP-7 O/S. In Linux, the VFS provides a uniform abstraction regardless of the underlying disk device.

For more detail on Linux file systems see Chapter 13

## Multitasking

Also akin to the early Unix, Linux borrows the basic multitasking operations of `fork` and `exec` to create and spawn “child processes”.

For more detail on the Linux Scheduler see Chapter 9

## Virtual Memory

Conceptually, the VM model used in Linux is a direct descendent of the paging model used in Multics and the CTSS.

Linux Memory Management is discussed in more detail in Chapter 11

## Kernel Space vs User Space

While not a Unix-ism per se, as mentioned before, an important feature of the Linux architecture is that it is a “monolithic” kernel which draws a hard distinction between processes within the kernel and those run in “user land”. In the converse “microkernel” design, the core is small and simple, and which interacts with all services through message queues. This allows the core design to be very small and to stay very clean, but also inflicts performance overhead for the managing of the message lists.

In the monolithic kernel, this overhead is by-passed by allowing kernel-space processes to share the same memory as the core kernel, and to communicate with the kernel and with other modules via direct C language function calls. As a result, any component in the kernel can communicate directly through the public functions any other component … providing of course that this hack will pass the scrutiny of the other developers.

That last item is the sole reason why Linux monolithic design does not degrade into a stew of madness: As with the C language itself, the “strictly controlled API” in the Linux architecture is not enforced by “bondage-and-discipline ethics”[2] in the tools, but by the self-discipline of programmers themselves. “Restraint when reasonable” is a cultural more that permeates the social, programming and architectural style in Linux.

# New Features in Linux

Charting Linux is like charting the migration of caribou or the vanishing cod stocks; you can tag a few, scour the tundra for tracks and make guesses, but you really can’t know where they are or where they are going until they actually get there. We can only now say what is actually in the 2.4 kernel, and what will be retained and revamped through the 2.5 path can only be stated as a wish-list of best intentions.

Of course, the future features of Linux is not entirely wishing upon a star. Reading this book is evidence enough that the future feature set can be a self-fulfilling prophesy, and, unlike similar books on any prior technology, the open source dimension of Linux means each and every reader of this book is granted full license to influence that future.

## Linux 2.4

If you are coming to this book from the 2.2 Kernel reference material, you're in for a ride. Systems have changed all over the place. The following is just a partial list of systems which have been added, re-written or rethought in the 2.4 kernel:

- Resource Management Subsystem
- Remapped Device Files
- Numerous changes to the `/proc` Interface
- Support for 4.2 billion users
- Memory limit is now 4G
- I2O/PCI improvements
- Improved Plug'n'Play Support
- Improved Parport Interface
- USB Support
- ia64, S/390 and SuperH Ports
- Shared memory VFS
- Logical Volume Manager
- Single Buffer File Caching
- Raw I/O devices
- Large-file file systems (file > 2G)
- UDF and UFS file systems
- SCSI Subsystem
- Module access control
- Network Address Translation (NAT)
- DECNet and ARCNet
- Direct Rendering Manager
- Voice Synthesizer Support
- Kernel-level HTTPD
- Kernel DocBook and **docgen**

## Planned features for Linux 2.5

Kernels take time to polish for production, and books take time to prepare and print; the net results are those features which were either not ready in time to be considered for the 2.4 kernel, and those features which were not well understood enough to be included in this book prior to press.

If you are developing in any of these areas, you would be well advised to tread carefully and to consult the `MAINTAINERS` file or the `linux-kernel` mailing list to get up to speed on the developments in these subsystems. Details on these and other enhancements bound for the 2.6 kernels are typically outlined on the linux-kernel TODO list (http://kena.8k.com/linux-kernel/TODO.txt) and should also be available from the usual kernel resources. (see Appendix B)

## CML2

You have probably configured your share of Linux kernels, and if you have yet to configure a 2.4 series kernel, you are in for a real treat. If you have ever tried to *add* a new kernel feature, you have already seen why Eric Raymond is spending his flight and hotel time redesigning the kernel configuration tools.

CML2 is the Configuration Modeling Language to succeed the current **make xconfig** method with a simpler system, easier to maintain and to program, and which will allow module and service developers to specify all dependencies of their code; it may even be possible to specify your hardware to the CML2 interpreter and to have it deduce the kernel options without intervention.

## System Calls

Several authors have called for removing or re-writing several of the core system calls. This list includes adding the `revoke()` system call, dropping `cli()` and `sleep_on()`, extending the utility of `setrlimit()`, `getrusage()`, and `wait4()`, and a re-write of `sysctrl()`.

## Frame Buffers (fbdev)

James Simmons has plans for cleaning up the fbdev layer with a new api, add support for cards with multiple frame buffers and add real multi-head support to the console system.

## Sound System

Although partially present in 2.4, the next kernel will expand the support for the ALSA sound drivers.

## V4L2

The 2.4 kernel contains little in the way of new support for the video and FM radio interface primarily because the developers are reworking the entire system for V4L2; unfortunately V4L2 was not

considered stable enough to be included into the 2.3 development tree and has been pushed back to Linux 2.5.

### Memory Management and Memory Devices

Rik van Riel has plans to support allocating very large continuous memory blocks on system initialization, and improved dcache and other improvements to the VM. David Woodhouse is also calling for adding support for flash and other memory device drivers.

### Journaling File System

Journaling file systems are a bit of a sore spot. The short story is that you can add one to any Linux 2.4 with very little effort, and some distributions have already bundled the ReiserFS with the 2.2 kernel releases. The bad news is that, to include a journaling file system, you will need to choose one, and a clear winner is not yet apparent. As a result, none of the options are included in the 2.4 kernel source tree.

The 2.5 TODO list includes merging in the changes for the ReiserFS and ext3, merging in XFS, HFS+, JFS and NWFS; whether all of these will happen or if some hybrid occurs remains to be seen.

## Tour of the Kernelbook

The chapters which in Part I in *The Linux Kernel* expand on the concepts in this chapter and take a closer look at the underlying architecture of the Linux kernel from a high-level view. A discussion of “The Reasonable CPU” describes the design constraints of the idealized processor which guides the portability requirements, followed by a general tour of the major kernel subsystems, the kernel source tree and a survey of kernel development and debugging tools. This section ends with a guide to participating in the Linux project, how to report bugs, contribute code and take full advantage of the open source revolution.

To create new services or to comprehend and adapt specific subsystems, you probably need only the detailed guide to the particular area and a higher-level view of its interactions with other components. The subsequent parts of this book step progressively deeper into the architecture to provide the detailed, domain specific views, first with a detailed subsystem by subsystem analysis of the core kernel in Part II and then a level deeper in the discussions of specific classes of kernel services in Part III in *The Linux Kernel*. The final part of this book, Part IV in *The Linux Kernel* is a collection of contributed chapters on advanced applications involving adapting the kernel.

There is no “magic bullet” to understanding the Linux kernel and while a book may be able to give you simple recipies to creating specific drivers and services, there is no substitution for reading the source; as

you follow through the chapters of this book, you should test all assertions against the source code and verify all potential bugs against current sources.

# Notes

1. Because of the GPL, Linux cannot fork into secretive proprietary factions: Any variants must make their variations known to the other communities and as a result, temporary diverging editions tend to quickly anneal. In open source development under a free sharing license, the best solution is sought in parallel by many independent developers, and results are compared and evaluated by the entire community. If there is a best solution, this methodology efficiently finds it. Also, although he holds the copyright to the Linux name, Linus Torvalds points out that, because he never bothered to acquire copyright transfer from the authors of all the patches he receives, it would be virtually impossible to achieve the consensus needed to switch the Linux source code from the GPL to another license model.
2. While used by many authors, this term is self-attributed to Eric Raymond who cites his `comp.lang.misc` posting from December 22, 1988 in response to the use of `:=` in Modula3:

   > … The European ‘bondage-and-discipline’ school of language design (the people who brought you Algol-68, Pascal, Modula, Ada, and Modula-2 and are now having yet another try at getting their mistakes right in Modula-3) likes to claim apostolic descent from that language …