Linux as a Case Study: Its Extracted Software Architecture

Ivan T. Bowman and Richard C. Holt
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
CANADA
+1 (519) 888-4567 x4671
{itbowman,holt}@plg.uwaterloo.ca

Neil V. Brewster
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 1A4
CANADA
+1 (416) 978-5036
brewste@cs.toronto.edu

ABSTRACT
Many software systems do not have a documented system architecture. These are often large, complex systems that are difficult to understand and maintain. One approach to recovering the understanding of a system is to extract architectural documentation from the system implementation. To evaluate the effectiveness of this approach, we extracted architectural documentation from the Linux^{TM} kernel. The Linux kernel is a good candidate for a case study because it is a large (800 KLOC) system that is in widespread use and it is representative of many existing systems. Our study resulted in documentation that is useful for understanding the Linux system structure. Also, we learned several useful lessons about extracting a system's architecture.

Keywords
Software architecture, architecture recovery, redocumentation

1 INTRODUCTION

Recent research [12, 15] suggests that large software systems should be designed with a documented software architecture. This architecture provides a building plan for a system at a high level of abstraction. Individual functions and even modules are not described in detail; instead, subsystems and relations between them are documented. This level of abstraction is appropriate for understanding an entire software system, and provides a good mechanism for system understanding.

We now know that using a documented software architecture throughout the lifetime of a software system can improve the quality and maintainability of the system. However, many existing systems do not have a documented system architecture. These systems are too valuable to discard or re-develop, but are often plagued by high maintenance costs, poor performance, or security risks. There is an approach that appears to be a promising way to get the benefits of a documented software architecture for these legacy systems: we can use automated tools to help *extract* architectural documentation from a system implementation. This approach has been used successfully by several researchers [3, 6, 10, 18, 20] to extract an architectural description from complex software systems.

Architectural redocumentation restores system understanding by abstracting important entities and their relationships in a large software system. This enhanced understanding can be used as part of a re-engineering effort, as a way to reduce maintenance costs, or as an input to a system evaluation. Unless architectural documentation is maintained, it will become obsolete as the system undergoes further changes. Finnigan et al. [3, 19] propose a way to keep architectural documentation up to date. First, automated tools are combined with human effort to extract system documentation and store it in a *Software Bookshelf*. As the system changes after the documentation extraction, a *librarian* uses automated tools to compare the system's implementation with the documentation. The librarian updates the documentation to reflect system changes (or perhaps prevents system changes that cause architectural erosion as described by Perry and Wolf [12]).

Linux^{TM} is a Unix^{TM}-like operating system that has received much popular attention [8]. Linux is based on the Open Source^{TM} concept [13], which means that there are no barriers to discussing the details of the system implementation. Linux has an interesting software structure that is similar to other large software systems. Linux is also a system that is growing rapidly; the source code for the Linux system has approximately doubled every year from 10 KLOC in 1991 to 1.5 MLOC in 1998 [9].

Because Linux is an interesting representative of existing software systems, we chose to examine it as a case study. In particular, we studied the Linux *kernel*, which is responsible for process, memory, and hardware device management. The Linux kernel is itself a large system (approximately 800 KLOC). Although there is some existing documentation about the Linux kernel [7, 14], this documentation describes individual subsystems and algorithms. There is no architectural documentation that describes the system structure at a high level of abstraction.

The Linux kernel is a good guinea pig for architectural recovery. It is a large system in widespread use, and it is an interesting representative of real software systems. Because Linux is freely available, there are no barriers to discussing its architectural structure in detail. To promote further use of the Linux kernel as a case study, we are making the architectural relations and system architecture that we extracted from the system implementation available to other researchers [11].

Case Study Approach

Two types of architectural documentation are particularly beneficial to humans trying to understand a software system: a *conceptual architecture* and a *concrete architecture*. The conceptual architecture shows how developers think about a system; it shows relationships between subsystems that are 'meaningful' to developers. For example, a subsystem might depend on another only for debugging purposes, but this dependency might not be shown in a conceptual architecture of the system. In contrast, the concrete architecture of a system shows the relationships that exist in the implemented system. While the conceptual architecture is easier to understand because it contains only essential relations, the concrete architecture is necessary when making decisions requiring implementation-specific knowledge.

Linux has neither a documented conceptual nor a documented concrete architecture. There is existing documentation that describes the kernel, but it describes individual subsystems in detail instead of concisely describing the relations between subsystems. As a first step in our recovery effort, we examined the existing documentation to determine the conceptual architecture of the Linux system. This conceptual architecture helped when examining the system implementation to form the concrete architecture—it allowed us to concentrate on important relationships, and provided an initial system structure.

Our approach to extracting the concrete architecture of the Linux kernel was as follows:

- Examine existing documentation to form a conceptual architecture of the Linux kernel.
- Group source files into subsystems based on directory structure, naming conventions, source code comments, and examination of the source code. Use the conceptual architecture as a guide in to what subsystems should be created and where files should be clustered.
- Extract relations between source files in the Linux implementation.
- Use the relations between source files and clustering of files to determine relations between subsystems.
- Use the clustering and relationships to form a concrete architecture of the Linux system.

Paper Organization

The rest of this paper is organized as follows. Section 2 describes the conceptual architecture of Linux. Section 3 describes the process we used to extract the Linux concrete architecture. Section 4 describes the concrete architecture. Section 5 draws conclusions from this work.

2 CONCEPTUAL ARCHITECTURE

We began our study of the Linux kernel by forming its conceptual architecture. This conceptual architecture acts as a framework which we use while examining the system implementation. The conceptual architecture helps us understand the volume of detail in the implementation by providing a suggested system structure.

We used descriptions [16, 17] of related operating systems (Unix and Minix) and existing Linux documentation to create an architectural description of the structure we expected to find in the Linux system. After reviewing existing documentation [7, 14], we arrived at the conceptual architecture, shown at its highest level of abstraction in Figure 1.

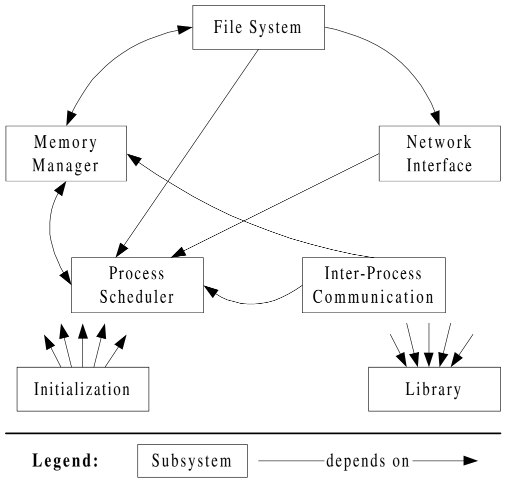

Figure 1: Linux Conceptual Architecture

The seven major subsystems are the following:

1. The *Process Scheduler* is responsible for supporting multitasking by changing which user process executes.
2. The *Memory Manager* subsystem provides a separate memory space for each user process, and uses swapping to support more processes than fit in physical memory.

3. The *File System* provides access to hardware devices. User processes can access keyboards, tape drives, hard drives, and modems using one interface that is implemented by the File System.
4. The *Network Interface* encapsulates access to network devices in a similar manner to the File System. User processes can communicate with other computers using several different types of network hardware and transmission protocols.
5. The *Inter-Process Communication* (IPC) subsystem allows user processes to communicate with other processes on the same computer. Synchronization, memory sharing, and inter-process messaging primitives are supported by the IPC subsystem.
6. The *Initialization* subsystem is responsible for initializing the rest of the Linux kernel with appropriate user configured settings.
7. The *Library* subsystem contains routines which are used throughout the kernel.

Each of the seven kernel subsystems has additional sub-subsystems hierarchically nested within it. The relationships shown in Figure 1 are 'depends-on' relationships. For example, the Memory Manager subsystem depends on the File System to swap memory to and from disk. For clarity, the relations from the Initialization subsystem and to the Library subsystem are omitted. The Initialization subsystem depends on all other kernel subsystems since it calls initialization routines throughout the kernel, and all of the kernel subsystems depend on the Library subsystem.

File System Conceptual Architecture

The Linux kernel is a large system that has a complex system structure. Its subsystems have sub-architectures of considerable size and complexity. Due to size limitations, we will focus on only one of these in this paper, the File System. Details of the other kernel subsystems are available in previous papers [1, 2].

There are three main roles that the File System performs:

1. It provides access to a wide variety of hardware devices.
2. It supports several different logical file system formats that control how files are mapped to physical locations on hardware devices.
3. It allows programs to be stored in several executable file formats, including interpreted scripts.

Figure 2 shows the conceptual architecture of the Linux File System. In this figure, double-headed arrows indicate dependence on or from all of the File System subsystems. This indicates that all of the File System subsystems depend on the Library subsystem, and the Initialization subsystem depends on all of the File System subsystems since it calls functions to initialize them.

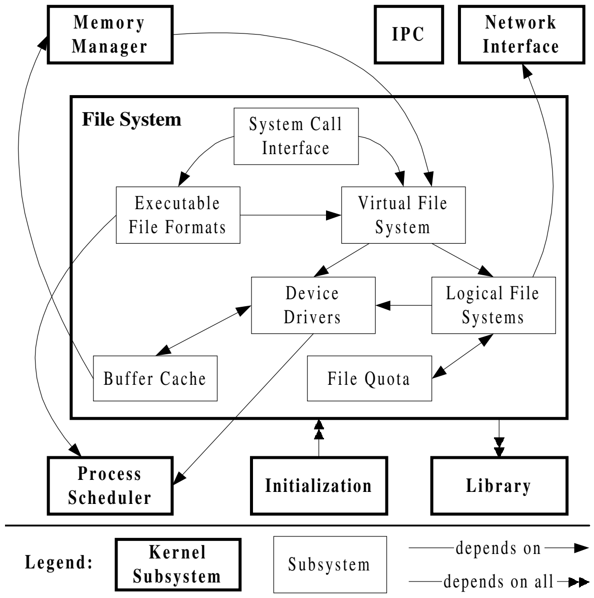

Figure 2: File System Conceptual Architecture

Linux uses the *facade* design pattern [4] to allow user processes and other parts of the kernel to use elements of the File System through a single interface. The facade design pattern uses a single subsystem which provides a single, simple facade interface to the subsystems within a system. Since clients only depend on the facade interface, the subsystems that implement system functionality can change their implementation without affecting clients. This design pattern allows clients to take advantage of a wide variety of hardware devices, logical file system formats, and executable formats without depending directly on any of the subsystems that implement specific functionality. Since user processes and other parts of the kernel depend only on the System Call Interface, subsystem interdependency is reduced substantially. The architecture of the Linux File System follows the 'object-oriented' or 'data abstraction' architectural style described by Shaw and Garlan [15]. The subsystems of the File System act to encapsulate state and functionality related to hardware devices or logical file systems. These subsystems interact through method calls.

The main roles of the File System are implemented in five subsystems.

1. The *Device Drivers* subsystem performs all communication with hardware devices supported by Linux.
2. The *Logical File Systems* implements several logical file systems that can be placed on hardware devices; these different file systems allow interop-

erability with different operating systems, and also allow specialized functionality such as encryption, compression, and high performance.

3. The *Executable File Formats* subsystem allows clients to execute programs from several different executable file formats, including not only compiled programs, but also interpreted scripts.
4. The *File Quota* subsystem allows system administrators to limit the amount of file storage that individual users may use.
5. The *Buffer Cache* subsystem provides memory buffers for input/output operations, and reduces hardware accesses by caching data and eliminating redundant reads and writes.

There are two other subsystems in the File System: the System Call Interface and Virtual File System subsystems. These two subsystems are facade interfaces. The Virtual File System combines functionality from the Logical File Systems and Device Drivers into a single interface. Other kernel subsystems can use the Virtual File System and treat all hardware devices as files, without depending on any particular hardware device driver or logical file system. The System Call Interface subsystem presents a similar unified interface. User processes can use the System Call Interface to access any functionality in the File System, without depending on the implementation of subsystems within the File System.

The dependency relations shown in Figure 2 are based on existing system documentation. Some of the dependencies that are perhaps unexpected are the following:

- The Device Driver subsystem depends on the Process Scheduler. While a hardware request is being completed, the associated device driver informs the Process Scheduler that the requesting user process should be suspended so that another process can execute.
- The Logical File System subsystem depends on the Network Interface subsystem. Three of the logical file system implementations represent files that are stored on another computer and accessed using the network. These three logical file systems depend on the Network Interface to communicate with the remote computer.
- The Memory Manager subsystem depends on the Virtual File System subsystem to swap memory to and from secondary storage.

In our conceptual architecture, no kernel subsystem depends on any particular File System Format subsystem, Device Driver subsystem, or Executable File Format subsystem. The Memory Manager subsystem is the only subsystem to depend on the File System, and it does so through the Virtual File System subsystem facade. Because of the facade design pattern, the Linux File System architecture is very flexible. It appears it would be easily maintainable because there are few dependencies.

The documentation we reviewed provided us with a conceptual architecture that indicates the Linux system is implemented according to strong implementation-hiding principles, and that the system should be easily understandable and maintainable. To find out whether the implementation matches this architecture, we need to extract a concrete architecture from the system implementation.

3 EXTRACTION METHODOLOGY

To determine what relations exist in the system implementation, we need to look at the definitive artifact—the system source code. The size of the Linux kernel implementation (800 KLOC) makes it too costly to examine the source manually. Instead, we used automated tools to extract relations from the source code then combined these relations into a concrete system architecture.

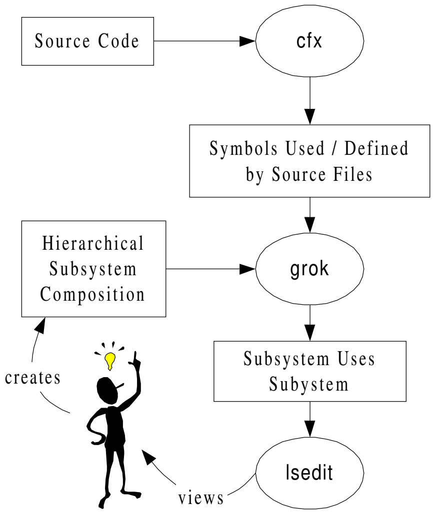

Figure 3: Extraction Process

Figure 3 shows an overview of the process we used to extract a concrete system architecture from the Linux kernel. We began by determining which source files were part of the kernel. Next, we used a source-code extractor called `cfx` [11]. This tool extracts relations such as 'function x calls function y' and 'source file x.c defines function x'. The tools extracts function call and variable access relations; these imply control flow and data flow dependencies.

The output of `cfx` is a set of relations between functions and variables. These relations are too detailed for human consumption. We used the `grok` [5, 11] tool to determine relations between source files based on the relations between the functions and variables defined within the source files. With 1682 source files in the kernel implementation, even relations between source files are at too low a level for easy system understanding. Instead of relations between source files, we would like to examine relations between subsystems. To achieve this result, we manually created a tree structured decomposition of the Linux system into subsystems. Each source file was manually assigned to a single subsystem, and each subsystem was assigned to a single containing subsystem. We used the subsystems from our conceptual architecture as an initial set of subsystems, and assigned source files to subsystems based on several criteria: directory structure, file naming conventions, source code comments, documentation, or, as a last resort, examination of the source code. If a set of source files seemed logically related, we created a new subsystem to contain them.

After we manually created a hierarchical description of the subsystems and source files in the Linux kernel, we used the `grok` tool to determine what relations exist between subsystems, based on the relations between the source files that are contained in the subsystems. The output of the `grok` tool is at the appropriate level of abstraction (inter-subsystem), but it is still difficult to understand directly. We used a visualisation tool called `lsedit` [5, 11] to visualise the extracted system structure. After viewing the extracted structure, we refined the hierarchical decomposition of the system by moving some source files to more appropriate subsystems.

Our extraction process combined tool support and human interpretation to extract the concrete architecture of the Linux kernel.

Hierarchical Decomposition

Before viewing the concrete architecture of the Linux kernel, we manually created a hierarchical decomposition of the system structure, assigning source files to subsystems, and subsystems hierarchically to subsystems. Figure 4 shows part of this hierarchical decomposition (some subsystems are omitted for brevity).

The seven major subsystems from Figure 1 are shown in the second and third rows of Figure 4. These subsystems also have corresponding directories in the source code implementation, which allowed us to quickly assign files within these directories to one of the major subsystems. Two of these major subsystems (the File System and Network Interface subsystems) had further subdirectories. Where possible, we used the directory structure to assign source files to appropriate subsystems. Where directory structure was not sufficient, we used file naming conventions and examination of the source code to place source files in subsystems. After applying these rules, we arrived at a tree-structured de-

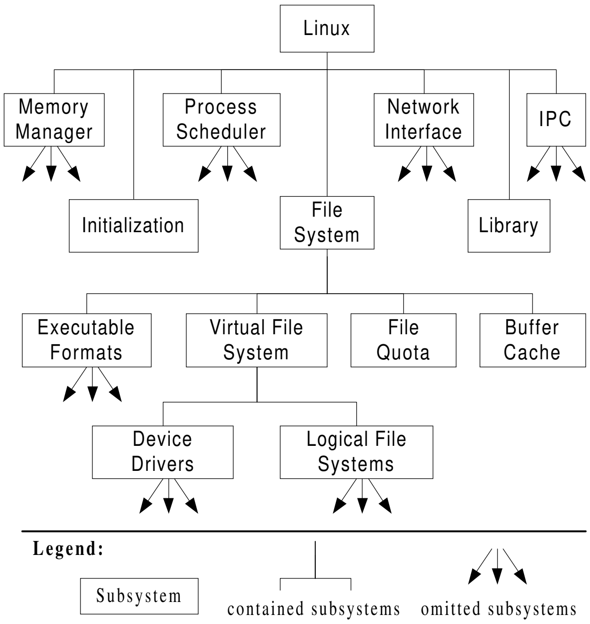

Figure 4: Partial Subsystem Hierarchy

composition of the Linux kernel such that each source file was placed in a single subsystem.

We used this hierarchical decomposition to view relations between subsystems instead of relations between source files. This level of abstraction made it possible for us to consider the structure of the entire Linux system.

4 CONCRETE ARCHITECTURE

A combination of automated extraction tools and human interpretation allowed us to determine the structure of the Linux kernel implementation. Figure 5 shows the relations that we found at the highest level of abstraction.

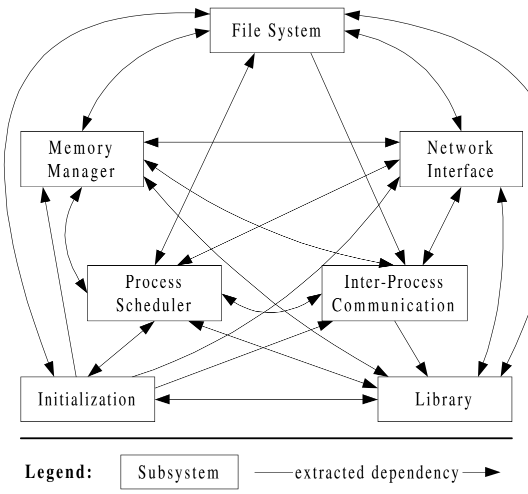

Figure 5: Linux Concrete Architecture

The concrete architecture in Figure 5 has the same subsystems as the conceptual architecture in Figure 1. However, the dependency relations appear to be quite different from the conceptual architecture. The conceptual architecture has relatively few dependencies between top-level systems with only 19 inter-subsystem dependencies. In contrast, the concrete architecture that we extracted is almost fully connected, with 37 inter-subsystem dependencies out of a possible 42.

When we examined these unexpected dependency relations, we learned that they appeared for several reasons. In some cases, Linux developers avoided existing interfaces for better efficiency; in other cases, it appeared that the dependencies appeared only for expediency. Whether or not these dependencies are required or desirable, we learned that a concrete implementation is likely to have more dependencies than a conceptual architecture indicates.

File System Concrete Architecture
To further compare the conceptual and concrete architectures of the Linux system, we examined the File System. Figure 6 shows the concrete architecture that we extracted from the File System subsystem.

The differences that we noted at the highest level of abstraction were also present within the File System. The concrete architecture of the File System has the same subsystems as the conceptual architecture, but there are substantially more dependency relations.

We studied the dependencies that appear in the concrete architecture but not in the conceptual architecture. Some we found to be quite surprising: the Network Interface subsystem depends on the Logical File System implementation directly, which we did not predict in the conceptual architecture. We found that two file systems (NCPFS and SMBFS) that use the network were implemented by having the Network Interface directly call functions in the implementation of these logical file systems. This is substantially different from our conceptual architecture, which predicted that the Network Interface would not depend on the File System at all, since network-oriented file systems would call the Network Interface to implement their functionality. From this dependency, we learned that unexpected dependencies can occur if control flow is implemented differently than expected.

Another dependency that we did not expect is the dependency of the Process Scheduler on the Device Driver subsystem. The Process Scheduler has a routine (`printk`) to print messages to the console. The `printk` routine calls a routine which is implemented within the Device Drivers subsystem of the File System. This dependency wasn't part of our conceptual architecture.

We found that all of the File System subsystems depended on the Inter-Process Communication (IPC) subsystem, contrary to the conceptual architecture prediction that none of these subsystems would depend on the IPC subsystem. Upon examination, we found that the IPC subsystem implements synchronization primitives that are used not only by user processes, but also by the rest of the kernel.

In addition to the above unexpected dependencies, we found that several dependencies could not be explained by an examination of the source and system documentation. It appears that these dependencies are due to developers avoiding existing interfaces for expediency.

Overall, the concrete structure of the File System is similar to the conceptual architecture that we formed based on available documentation and related systems. However, we found that there were substantially more dependency relations, caused by missed dependencies in the documentation, functionality that was implemented in multiple subsystems, and unexpected control flow implementations. Although there are substantially more dependencies in the concrete architecture of the File System than the conceptual architecture, the system is still far from fully connected. It appears that it is not as easy to maintain and update the File System as the conceptual architecture indicates, but the File System still appears flexible and open to change.

Logical File System Concrete Architecture
To further explore the concrete architecture of the Linux kernel, we examined the Logical File Systems concrete architecture. The Logical File Systems subsystem contains seventeen different logical file systems. These file systems are responsible for mapping logical files (which are presented to user processes through the System Call Interface) to physical locations on storage devices. Figure 7 shows the concrete architecture of the Logical File Systems subsystem.

In the conceptual architecture in Figure 2, we predicted that there would be a separation between the interface to logical file systems and their implementations. We expected that other kernel subsystems would not depend directly on any implementation of a logical file system, instead depending only on the Virtual File System subsystem. This separation follows the facade design pattern [4]. When we extracted relations from the system implementation, we found that the situation was not so simple.

One set of dependencies that we did not expect are due to the PROC file system. This file system is a special file system that reports status information about the kernel, and allows access to the status and memory of executing processes. To accomplish this reporting, the PROC file system relies on other kernel subsys-

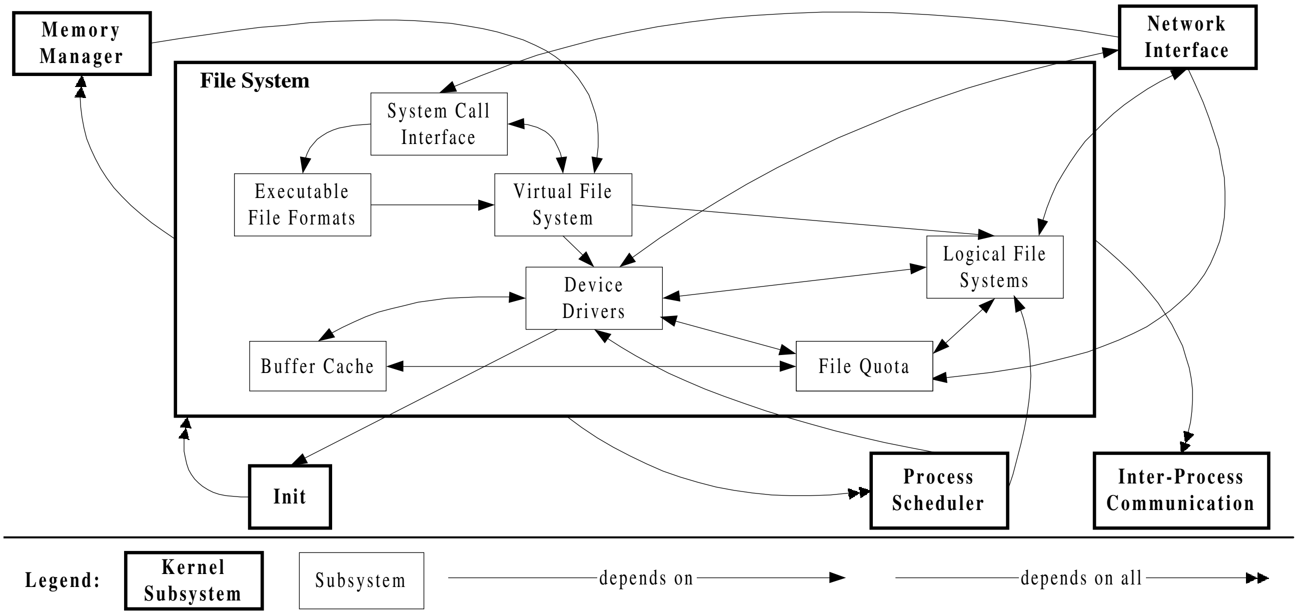

Figure 6: File System Concrete Architecture

tems to perform reporting about their status. Because the reporting functionality is implemented throughout the kernel, the Process Scheduler and Network Interface subsystems depend on the PROC file system.

Another dependency that we did not expect is the dependency of the ISO subsystem on the CD-ROM device driver. We had expected that logical file systems would not depend on any particular device driver implementation, instead depending only on the Facade Interface of the Device Driver subsystem. We found that the ISO9660 logical file system is only used on CD-ROM devices, and there are data types that are defined by the CD-ROM device driver and used by the ISO9660 file system.

We did find that the different Logical File Systems are relatively independent of each other. The exceptions are those systems that reuse code: the SMBFS, NCPFS, FAT, VFAT, UMSDOS, and MSDOS subsystems implement access to various MS-DOS^{TM} related file systems. Because these subsystems share functionality, they reuse code. This reuse leads to dependencies between the subsystems. The implementation of the EXT2, XIA, SYSV, EXT, and MINIX file systems is based on similar reuse, again leading to unexpected dependencies.

The Logical File Systems subsystem of the Linux File System has more dependencies than we had predicted in our conceptual architecture. In addition, different Logical File Systems are not isolated from each other to the extent that we had expected based on system documentation. However, the facade design pattern is apparent in the extracted system structure, and it appears to be relatively easy to add more logical file systems or update existing ones.

5 CONCLUSIONS

In our study, we used existing documentation and knowledge of related systems to form the conceptual architecture of the Linux system. Next, we used automated tools and human interpretation to extract the concrete architecture of the Linux kernel.

The conceptual architecture of the Linux kernel contains abstractions (such as the facade design pattern) which appear to limit inter-system dependencies and promote maintainability and extendibility. Although we were able to find these abstractions in the concrete architecture, we found that there were unexpected dependencies at all levels of abstraction. These extra dependencies act to reduce the maintainability of the Linux kernel. As the system grows, it is possible that these dependencies will need to be eliminated.

Lessons Learned

Our extraction effort showed us that automated tools are very helpful in extracting the architecture from a system's implementation. Our tools automatically extracted facts, and showed us relations at any level of abstraction that we wanted. However, we still needed a human's judgement to determine an appropriate hierarchical decomposition of the system structure based on idiosyncratic details such as directory structure, file naming conventions, and examination of source code.

We found that the concrete architecture of the Linux

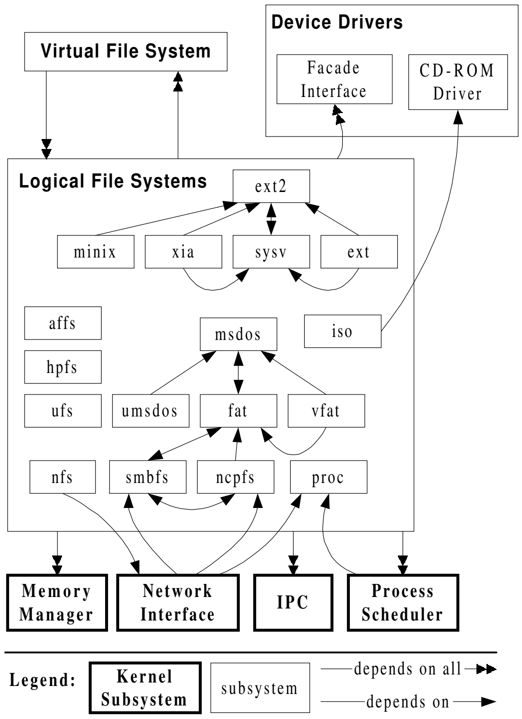

Figure 7: Logical File System Concrete Architecture

kernel has substantially more dependencies than the conceptual architecture. In fact, the Linux kernel is almost completely connected at the highest level of abstraction. We found the following reasons for additional dependencies:

- The conceptual architecture missed the use of some subsystems; for example, the IPC subsystem implements synchronization primitives that are used throughout the kernel, but the conceptual architecture shows the IPC subsystem used only by user processes.
- Some functionality that the conceptual architecture showed in a single subsystem was implemented in several subsystems, leading to additional dependencies. For example, the PROC file system is implemented throughout the kernel.
- The conceptual architecture might show control flow in one manner, but the implementation might use a different mechanism. For example, the conceptual architecture showed that the network-oriented file systems depended on the Network Interface. In the concrete architecture, we found that the Network Interface directly calls two of these logical file systems.
- In some cases, Linux developers improved system efficiency by bypassing existing interfaces.

In addition to the above reasons for additional dependencies, it seems that some of these dependencies exist for developer expediency. One comment in a header file states "The read-only stuff doesn't really belong here, but any other place is probably as bad and I don't want to create yet another include file."

The Linux system could be restructured to remove some unexpected dependencies. One thing that seems to have affected the use of implementation details is the organization of the source code: most of the header files that define system details are located in a single directory. Thus it is difficult to determine which header files define interfaces that should be used throughout the kernel, and which header files define interfaces for use within a single subsystem. In some cases, the placement of the header files is required by the implementation technique: the super-block of the virtual file system contains a union of information for each of the different logical file systems. This means that the file system (and any module that uses it) needs to have knowledge about the details of the implementation of each of the logical file systems.

After reviewing the concrete structure of the Linux kernel, it would be possible to update the conceptual architecture. Some dependencies in the conceptual architecture that we formed based on documentation were missed by simple omission—we did not see mention of the dependencies in the documentation, nor do they appear in related systems. The concrete architecture should be used to refine the conceptual architecture, but it is not desirable to add all relations from the concrete architecture since many of these relations are not essential, and hinder system understanding because of the additional complexity. For example, the dependence of the Process Scheduler on the Device Drivers subsystem of the File System through the single call in the implementation of `printk` could be omitted from the conceptual architecture. The development of the conceptual and concrete architectures seems to be best accomplished with an iterative process.

Although the structure of the Linux system is desirable in many cases because of efficiency or other considerations, it is likely that many unnecessary dependencies could be eliminated if the system was restructured to avoid using implementation details directly. It may not be reasonable to do this with Linux at this point, but perhaps when new systems are implemented, automated tools such as those used in this case study can detect and prevent these spurious relations.

ACKNOWLEDGEMENTS

The authors would like to thank Gary Farmaner for his support with the extraction tools. We would also like to thank Meyer Tanuan and Saheem Siddiqi for their help in extracting an earlier version of the architecture. Susan Sim provided valuable feedback on an earlier draft

of this paper. The contribution of the Linux developer community is gratefully acknowledged.

REFERENCES

[1] Ivan Bowman. Conceptual architecture of the linux kernel. Available at http://plg.uwaterloo.ca/~itbowman/CS746G/a1/, 1998.

[2] Ivan Bowman, Saheem Siddiqi, and Meyer Tanuan. Concrete architecture of the linux kernel. Available at http://plg.uwaterloo.ca/~itbowman/CS746G/a2/, 1998.

[3] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, October 1997.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[5] R.C. Holt. Structural manipulation of software architecture using tarski relational algebra. In *Eighth Working Conference on Reverse Engineering (WCRE'98)*, October 1998. To appear.

[6] Rick Kazman and Jeromy Carrière. View extraction and view fusion in architectural understanding. In *5th International Conference on Software Reuse*, Victoria, BC, Canada, June 1998.

[7] The linux kernel hacker's guide. Available at http://www.redhat.com:8080/HyperNews/get/khg.html.

[8] Josh McHugh. Freeware children. *Forbes Magazine*, August 1998.

[9] Josh McHugh. Linux: The making of a global hack. *Forbes Magazine*, August 1998.

[10] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, Washington, DC, October 1995. IEEE Computer Society Press.

[11] PBS tools. Available at http://www.turing.cs.toronto.edu/pbs.

[12] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[13] Eric S. Raymond. The cathedral and the bazaar. Available at http://sagan.earthspace.net/~esr/writings/cathedral-bazaar/, 1997.

[14] David A. Rusling. The linux kernel. Available at http://sunsite.unc.edu/Linux/LDP/tlk/tlk.html.

[15] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall Press, April 1996.

[16] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley, 5th edition, 1997.

[17] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[18] Vassilios Tzerpos and R.C. Holt. A hybrid process for recovering software architecture. In *Proceedings of CASCON 1996*, Toronto, Canada, November 1996.

[19] Vassilios Tzerpos, R.C. Holt, and Gary Farmaner. Web-based presentation of hierarchic software architecture. In *Workshop on Software Engineering (on) the World-Wide Web*, Boston, May 1997. International Conference on Software Engineering 1997.

[20] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 11(6):501–520, January 1995.