Concrete Architecture of the Linux Kernel

Moataz Kamel[1]
James P. D. Keast[2]
Chris Pal[2]
m2kamel@swen.uwaterloo.ca
james@keiths.uwaterloo.ca
cpal@csg.uwaterloo.ca

12 February 1998

[1] Department of Electrical and Computer Engineering
[2] Department of Computer Science
University of Waterloo

at
Waterloo, Ontario, N2L 3G1

ABSTRACT

This report presents the concrete, as implemented, software architecture of the Linux kernel. Linux is a UNIX like operating system written by Linus Torvalds. The C language source code for Linux 2.0.30 contains over 800 KLOC and as such is a good candidate for analysis. The architecture was extracted from the source code through a reverse engineering process using the Rigi system from the University of Victoria [10]. The architecture presented provides detailed enough information about the Linux kernel to be useful to someone tasked with maintaining or expanding the kernel.

Keywords: Linux Kernel Architecture, Rigi, Reverse Engineering.

Report prepared for **Computer Science 746G.** ***Software Architecture.***
Instructor: Ric Holt

Contents

List of Figures

1 Introduction

1.1 Purpose

This report provides a description and discussion of the concrete, as implemented, architecture of the *Linux*[*] operating system kernel. The concrete architecture serves to illustrate the actual subsystem and module decomposition that was implemented in Linux 2.0.30. This can be compared to the conceptual architecture in order to asses the correspondence.

In this report, modules are defined to be C language source files. They have been grouped together into subsystems based on the functionality they provide. For the most part, this subsystem grouping corresponds to the directory structure of the Linux source code.

The primary purpose of extracting the concrete architecture from the Linux kernel was to gain experience with this sort of reverse engineering process and the notion of architecture extraction in general. The result of this extraction is the concrete architecture presented here. This architecture would be useful for anyone who wished to learn more about how the different parts of the Linux Kernel interact at the subsystem and module level. In particular, it would be of value to anyone who wanted to expand or maintain the Linux Kernel.

1.2 Conceptual Architecture

Figure 1 shows the conceptual architecture of the Linux Kernel as presented by Keast in [4]. The conceptual architecture focuses on the conceptual organization of the system and leaves out details of the implementation. The authors views of the conceptual architecture of Linux can be found in [3, 4, 6]. This figure is reproduced here merely to provide a context for the concrete architecture presented in this report.

1.3 Background Readings

Background understanding of the Linux kernel was acquired from a number of online document sources. *The Linux Documentation Project* [1] world wide web page is particularly useful in providing links to technical documents which cover all aspects of the Linux project. Two documents which provide excellent coverage of the Linux kernel are the *Linux Kernel Hacker's Guide* [2] and the online version of the book *The Linux Kernel* [7]. Finally, a useful reference is the Linux Kernel Tour provided online by Tama Communications[11]. This tour was useful for verifying which modules called which modules and for confirming the information extracted by Rigi.

1.4 Notes on Rigi

The Rigi system[10] is a software tool which was used to extract information from raw source code and to organize this information graphically into higher level abstractions in order to arrive at the architecture of the Linux Kernel system. To understand the figures in this report, it is important that some of the conventions of Rigi be explained.

[*] *Linux* is a trademark of Linus Torvalds.

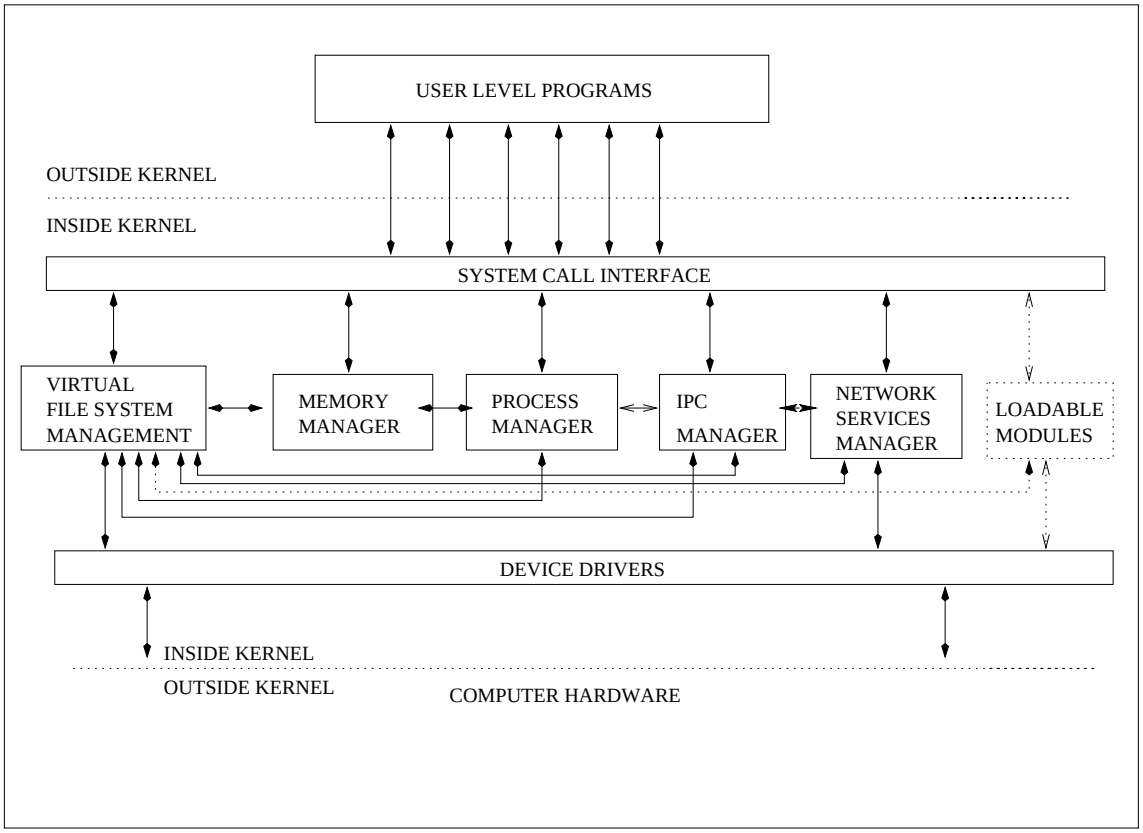

Figure 1: Conceptual architecture of the Linux Kernel[4].

Rigi represents software artifacts such as functions and data structures as boxes. Groups of these artifacts can be selected and "collapsed" into a single box representing a module or subsystem. By convention, Rigi indicates dependency on a module or subsystem as arcs entering the box from the top. Likewise dependency of a module or subsystem on other modules are indicated by an arc exiting the box from the bottom.

Rigi makes use of different colored arcs to represent different relationships. Red arcs indicate call dependencies while yellow arcs indicate data dependencies. Green arcs are "composite" arcs and encapsulate several other colors or dependencies. Rigi also color codes boxes with green for functions, yellow for data, and orange representing aggregates (i.e. modules and subsystems).

Several problems were encountered when using Rigi to extract the Linux architecture. In most cases, these problems can be attributed to errors in how Rigi interpreted the sometimes bizarre coding structures used in the kernel. For example Rigi was confused by the use of macros. Often macros were interpreted as functions or procedures and the figures produced by Rigi suggested that a module was calling many modules when in fact many modules were making use of a macro defined in one module. Another common problem was that Rigi could not distinguish between symbols with the same name but that were defined/implemented in multiple files. This problem resulted in Rigi showing dependencies that did not really exist in the code.

In general, it was necessary to manually review the dependencies that Rigi generated and to verify their correspondence to the source code. In order to accomplish this for a system the size of the Linux kernel would require many weeks of dedicated work. Although some attempt was made to correct inaccuracies in the extracted information, not all of these were removed and thus some of the information contained in the figures may still be inaccurate.

1.5 Organization of this report

This report is organized as follows. Section 2 contains an overview of the architecture of the entire system and its subsystems. Next a detailed examination of each subsystem, based on the implementation, is provided in Section 3. The interaction among the kernel's subsystems is also described in this section and the structure of each of subsystem is elaborated. The architecture description is supported by figures generated with the Rigi system. A data dictionary of key terms is provided for reference in Appendix A.

2 High Level Structure of The Linux Kernel System

2.1 Design Patterns

Linux was developed by many people around the world. Due to this geographic separation, the internet has been an invaluable tool for the development of Linux. For this and other reasons, Linux has a intimate relationship with networking and distributed computing. Thus we will define two views of kernel architecture with terminology from distributed systems research. In the past, researchers have traditionally defined two architectural patterns concerning distributed systems.

> One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from user-level servers. These two models are known as the monolithic kernel and the microkernel.[9]

From an abstract viewpoint, the Linux Kernel is patterned as a monolithic kernel design, as most of the kernel services run in kernel mode not user mode. However, developers soon realized that space was easily wasted if unused code was left in the kernel. In the classic monolithic architectural style, this code could only be removed by recompiling the kernel. In this spirit, the Linux 2.0.30 source code contains a small subsystem known as menuconfig which is essentially a simple GUI allowing the user to configure the kernel and recompile the kernel with different features. However, this can be a bit of an inconvenience. Thus, the Linux community also has defined another way to change the configuration of the kernel, in terms of a *run-time linkable module* that can be loaded at runtime. These *run-time linkable modules* consist primarily of drivers and file systems. There have been some rough guidelines illustrated in the linux literature, however the process of adding a module and the specific interfaces to the rest of the kernel is quite variable depending on that module.

2.2 High Level and Low Level Components

Most conceptual Linux subsystems can be viewed in terms of two concrete components, an abstracted component and a processor specific, platform specific or device specific component. Of course, even the abstracted level must be aware of some low-level hardware detail. However, the distinction is that modules at the abstracted level are initialized with hardware parameters at the startup of the system. Further communication and direct knowledge of hardware is then obtained through lower level components. Many of these modules together in a subsystem define the abstract component of that subsystem. The source code for these modules is usually located in the directory structure at the /sybsystem/ (e.g. /mm/ or /kernel/) area just down from the root. The platform specific modules are much more aware of the hardware during execution, often containing assembly level

code. Together these modules form the platform specific components of a subsystem. These modules are usually located in the `/arch/(i386, Alpha, m68k)/(mm, kernel)` area of the source code. Modules interact within themselves and with one another at both of these levels. Additionally, conceptual modules can interact with the abstracted components of each other through intermediate calls involving platform specific procedures. Thus, interactions between conceptual subsystems can be characterized in terms of purely abstracted interactions and processor specific interactions. The source code analyzed in this report included code for Digital Alpha, Intel 386, Motorola 68000, MIPS, Power PC and SPARC platforms.

2.3 The Init Subsystem

Init is a small subsystem that takes over once low level code has dealt with the booting up procedure. Within Init are a small number of simple procedures that are used by other modules. These procedures consist mainly of simple low-level tasks such as printing to the screen. Init makes some simple calls that parse any command line arguments and initialization scripts affecting the configuration of the kernel. Init then communicates with all the other modules to initialize each of them. The sequence of actions taken by the initialization procedure also sketches a view of the major subsystems in the kernel. When the init module has finished initializing the other modules in the sequence listed below, it creates a kernel thread that will assume control when the system is not busy.

1. A second CPU is initialized as the slave if there are more than one defined
2. Hardware specific initialization is performed
3. The paging system is initialized
4. Traps are set with a platform dependent call
5. IRQ's are initialized with another platform dependent call
6. The scheduling system is initialized
7. Modules are initialized based on command line arguments
8. Numerous features of the abstract virtual file system are initialized
9. The networking module is initialized
10. The inter-process communication module is initialized

2.4 The Memory Manager (MM) Subsystem

The Memory Manager subsystem is responsible for maintaining the structure of the memory in the system, monitoring ownership of memory among the process of the system and managing virtual memory (e.g. handling page faults for demand paging which was implemented on 01.12.91 [5]). Platform specific settings configured when the system is initialized allow routines in the memory manager to handle hardware page faults when they occur later at the platform specific level. Page Faults handled at the low level of the MM cause communication with the abstract component of the kernel to determine information on the process originating the fault (e.g a call `force_sig`). Communication also occurs if a process is swapped out. The Memory Manager makes extensive use of two data structures: the main memory manager data structure (`mm_struct`) and the virtual memory

structure (`vm_struct`). Communication with these structures usually occurs the abstracted level of the MM, however it may also occur at the platform specific level. For example some platform specific configurations will interact with these data structures from lower level code. The Memory Manager communicates with the File System through shared data and procedure calls. Two main data structures are used in the communication, the swap information structure and the I-node structure. Numerous procedures involved with the paging and buffer management system make calls to various File System procedures. The MM also communicates with the Networking System when gathering information on the status of buffers used by the Networking System. The MM communicates with the IPC subsystem usually when a process is about to be swapped out.

2.5 The Kernel Subsystem

The abstract layer of the Kernel is responsible for a number of duties mostly related to managing processes, such as: the scheduling of processes, creating new processes, terminating processes, processor signals (e.g. invalid instruction), direct memory access (DMA) access management, loading modules and reserving certain kernel resources. The platform specific component of the Kernel is responsible for such duties as: interrupt management, low level process creation and termination, system shutdown, BIOS interaction and other lower level duties. In older versions of Linux support for math co-processor emulation was also provided at this level. The Init subsystem communicates with the platform specific component of the Kernel when initializing multiple CPU's (Intel and Sparc only). Additional communication occurs when initializing the Kernel configuration based on processor specific procedure calls determined by the compiled configuration of the overall system. These initialization calls cause the platform specific layer of the Kernel to communicate with the low level of the Memory Manager in order to setup platform specific memory addressing information. After the Initialization process, the abstracted level of the Kernel communicates with the abstracted level of the Memory Manager with procedure calls for three main reasons. It calls procedures in the memory manager to:

1. allocate memory for modules that are loaded and processes that are forked,
2. gather system memory usage information, and
3. verify ownership of various portions of memory during numerous system calls (e.g. `sys_signal`).

However, other system calls originating at the lower level of the Kernel (e.g. `sys_ptrace`) also call procedures at the abstraction layer of the Memory Management subsystem. Communication with the Memory Manager is primarily through the important procedure `verify_area` at both the abstract and low level of the Kernel. This is a highly optimized procedure in the abstraction layer of the Memory Manager that prevents processes from interacting with one another by mistake and enforces other aspects of memory ownership. It is:

```
/*
 * Ugly, ugly, but the goto's result in better assembly..
 */ [mm/memory.c]
```

Communication between the abstract layer of the Kernel and the File System usually takes the form of procedure calls to the File System and are usually involved with mounting files systems. Additional communication occurs when a program is invoked by the user and executable code is pulled from the File System into memory. Communication to the Networking subsystem at the platform specific layer is minimal but some processor specific builds allow the architecture specific layer of the Kernel to communicate directly with the Networking subsystem to initialize sockets.

2.6 The Inter-Process Communication (IPC) Subsystem

Linux supports classic System V inter-process communication including: semaphores, shared memory and message queues. The IPC subsystem is responsible for managing these communication modes between processes. Whenever a system call is made for IPC they are de-multiplexed by the platform specific layer of the IPC subsystem (shown below).

```
/*
 * sys_ipc() is the de-multiplexer for the SysV IPC calls..
 *
 * This is really horribly ugly.
 */ [arch/sys_i386.c]
```

After de-multiplexing, a call is made to the abstracted layer of the IPC. This in turn invokes a call to the memory manager to verify memory in a similar manner as system calls originating in the Kernel. Data is shared with the Memory Manger in the form of the virtual memory data structure (`vm_area_struct`). This structure is accessed by procedures involving shared memory. The abstracted layer of the IPC subsystem communicates with the abstracted layer of the Kernel primarily when a message is being handled and the scheduler is notified with a procedure call to the Kernel (e.g `real_msgsnd` calls `wake_up`). When semaphore operations are invoked (e.g. `sys_semop` calls `interruptible_sleep_on`) a similar abstract layer to abstract layer interaction occurs. The IPC subsystem talks to Drivers and the Networking subsystem primarily during routines dealing with the CD-ROM device driver and other drivers for mass storage devices.

2.7 The Driver Subsystem

Various drivers communicate with the Memory Manager through requests for kernel memory (kmalloc). Other requests are also made for the popular reason of verifying ownership of memory. Data is shared with the Memory Manager in terms of page descriptors. All drivers communicate with the abstracted layer of the Kernel for io-region resource management (e.g with calls such as `request_region`). Some drivers may also utilize calls to the abstract kernel for direct memory access management or for requesting timing services (e.g. `add_timer`). The Kernel thus handles the resource allocation and management of *disk like* drivers used commonly by the File system. Calls from drivers to the file system usually involve the management of buffers and I-nodes through a simple interface defined in the abstract layer of the File System. In contrast Network drivers have much more communication directly with the abstract layer of the Networking subsystem when initializing and requesting resource management (e.g. calls such as `dev_alloc_skb`).

2.8 The Networking (NET) Subsystem

The Networking subsystem is a large system that provides networking communication to the processes running on the operating system. To accomplish this task, it communicates with the other kernel components in the following ways. Communication to the abstract layer of the Kernel subsystem often occurs in term of requests for various timed operations (e.g. through `add_timer` type calls). Extensive communication occurs when Unix socket connections are initialized or are idle. Requests to the Kernel subsystem are made to sleep processes managing the socket connection. For example: `sock_alloc_send` calls `interruptible_sleep_on` within the abstract layer of the Kernel subsystem. The abstract component of the Networking subsystem communicates with the abstract component of

the Memory Manager to allocate kernel memory and to verify memory ownership. Communication occurs with the File System when socket connections are manipulated and initialized since sockets, like files, are associated with I-nodes in the file system. Numerous I-node procedures in the abstract component of the File System are called from within the abstract levels of the Networking system. Data is shared with the rest of the system as files data structures (a number of I-nodes). A number of low level platform dependent calls directly to low levels of the Memory Management subsystem are made within the TCP/IP protocol suite located within the abstract component of the Networking subsystem. Extensive communication between the abstract layer of the Networking subsystem and the Drivers occurs as networking buffers used by the devices are removed, added and manipulated.

2.9 The File System (FS) Subsystem

The File System is responsible for manipulating data in terms of files that are usually stored on disks. Disks can be located on the local machine or remotely over the network. Thus as one might expect numerous procedures within the abstract component of the File System communicate with the abstract component of the Networking System (i.e Linux supports the Networked File System or NFS). This interaction originates from procedures in the File System that deal with NFS (e.g `root_nfs_open` calls `sys_socket`). The File system communicates with the Memory Manager to request allocation of kernel memory and to verify memory ownership during various system calls. The File System extensively manipulates the I-node hash table data structure. Demand paging is implemented in Linux, thus a large amount of interaction occurs between the Memory Manager subsystem and the File System subsystem to deal with the loading and writing of pages from and to disk. When programs are initially invoked and brought into memory from disk the File system calls a number of procedures in the MM subsystem to set up the program for execution in memory (e.g. `setup_arg_pages` calls `insert_vm_struct` within the abstract component of the Memory Manger).

3 Detailed Structure of the Linux Kernel Subsystems

3.1 Summary of High Level Kernel Subsystem Interactions

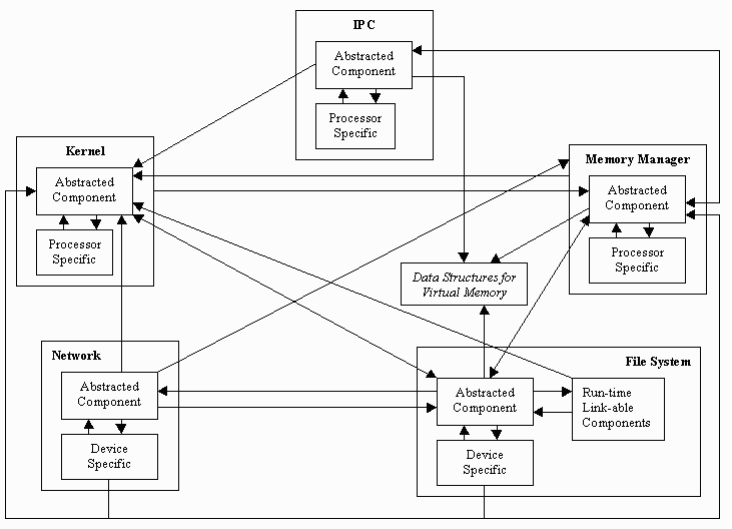

Figure 2: High level view of the Linux kernel's concrete architecture.

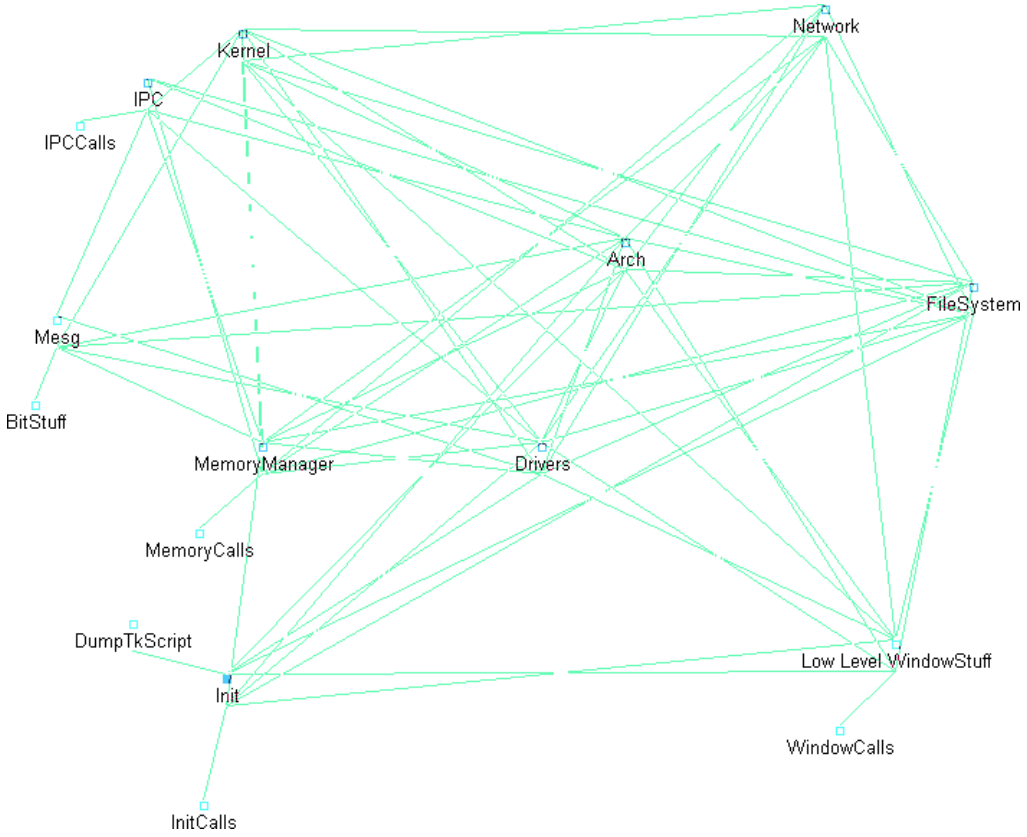

Figure 3: High level view of the Linux kernel's concrete architecture as extracted by Rigi.

The interactions among the various high level subsystems described above was extracted from the source code passed through RIGI and refined into subsystems. Most subsystem identification was made based on directory location and in some cases highly interconnected clusters of code were grouped into smaller subsystems. The preceding section was derived mainly though the identification of calls that were made between modules.

Figure 3 shows a high level representation of the Linux kernel as extracted using the Rigi tool. The figure is not easy to follow since due to the abundance of interactions between subsystems which appear as crossing lines. Figure 2 shows a simplified block and line diagram of the same architecture.

The interactions extracted from the RIGI system are indicated in the following manner: If an arrow leaves a box, all the items within the box call procedures within the target of the arrow. Similarly, if an arrow arrives at a box, the originator of the arrow may call anything within that box. Using this convention, interactions between the various components is rather straightforward. In general, for any given arrow there will exist a description of the interaction indicated by that arrow in the preceding section organized by the orginator of the call. For example, the File System calls procedures in the Networking system vice versa. This interaction occurs at the abstracted layer. This type of interaction was described in the previous section and consists mainly of calls dealing with NFS. In contrast the Kernel subsystem calls the abstract layer of the Memory Manager from both its abstracted component and from its platform specific component.

3.2 The Init Subsystem

Figure 4 shows the Init subsystem. This subsystem is where everything gets started. This is where all initialization and startup procedures takes place. The `main.c` module of the Init subsystem is the first module executed when the system boots. The primary module of this subsystem is clearly the `main.c` module. No modules call this module and it calls every other module. The graph produced by Rigi was very messy for this subsystem. In order to make sense of it, all modules except for `main.c` were placed into groups based on the general functionality they performed. The groups are explained below.

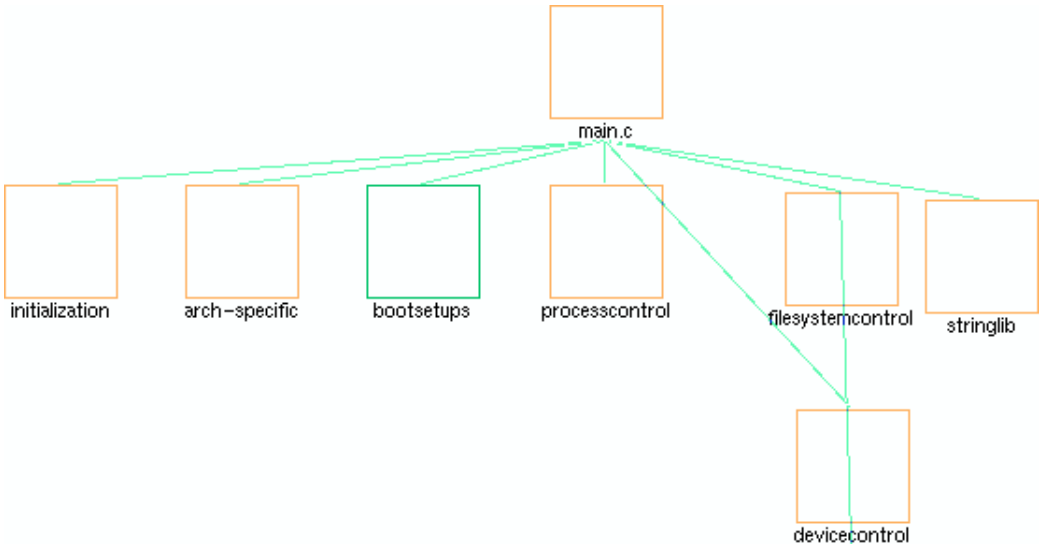

Figure 4: The architecture of the Init subsystem as extracted by Rigi.

- The `initialization` group contains all the modules which provide the various setup and initialization functions that `main.c` needs to call in order to get things going. This includes things like the initialization of the loadable modules, IPC mechanisms, memory manager, scheduler, file system manager, and kernel interrupt and trap handling.
- The `arch-specific` group contains the functions which `main.c` calls in order to perform any architecture (i.e. computer hardware) specific initialization.
- The `bootsetups` group contains the functions called by `main.c` to boot the kernel. This process involves loading the compressed Linux kernel into memory when the system boots.
- The `processcontrol` group contains the functions called by `main.c` to setup and manage processes and process threads.
- The `filesystemcontrol` group contain the functions called by `main.c` to initialize and manipulate the file systems.
- The `strignlib` group contains string manipulation functions used by `main.c` when performing the initialization.
- The `devicecontrol` group contains the functions called by `main.c` to perform device and device driver setup. For example there are calls to `ide_setup` (setup IDE devices), `check_tbl` (check various file system tables) and `check_fpu` (verify existence of floating point unit).

3.3 The Kernel Subsystem

Figure 5 shows the Kernel subsystem. This subsystem is responsible for such things as process scheduling, process forking, signal processing, interrupt handling and context switching. This subsystem also provides services used by external modules. For example the `panic()` function is provided in `panic.c` which allows modules in other subsystems to initiate a kernel panic.

The figure shows all the modules in the subsystem and how they are connected by calls. The figure also shows the global types (data structures) shared by the modules in the subsystem. In this case the arcs to these modules have been removed to improve readability. A list of types used by each module is given below.

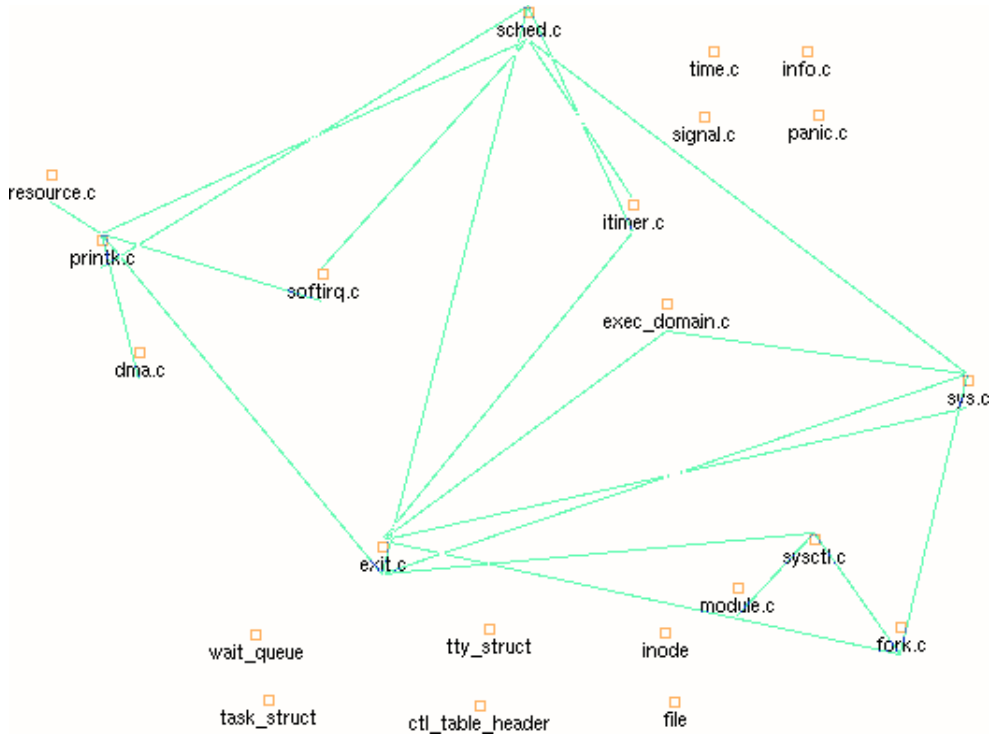

Figure 5: The architecture of the Kernel subsystem as extracted by Rigi.

The main module of this subsystem is the `sched.c` module which is responsible for scheduling all events within the kernel during runtime.

- The module `sched.c` is called from the `printk.c`, `itimer.c`, and `exit.c` modules. The calls from `printk.c` deal with handling of interrupt requests. The calls from `itimer.c` deal with the creation, resetting and deletion of kernel timers. The calls from `exit.c` deal with handling interrupt requests and making scheduling requests. This module uses the types `task_struct` and `wait_queue`.
- The module `dma.c` is not called from any of the modules in this subsystem. It provides functions for getting information about, requesting and freeing DMA channels.
- The module `exec_domain.c` is not called from any of the modules in the kernel subsystem. It provides functions for registering, unregistering, and looking up kernel exec information.

- The module `exit.c` is called by the modules `exec_domain.c`, `fork.c`, `itimer.c` and `sys.c`. These modules make calls into `exit.c` in order to end or reset processes and to reset the entire OS (`ctrl_alt_delete`). This module uses the type `wait_queue`.
- The module `fork.c` is not called from any of the modules in the kernel subsystem. It makes use of the type `task_struct`. It provides functions for forking new processes, assigning process ids and managing tasks (or threads).
- The module `itimer.c` is called from only the `sched.c` module. It provides functions which provide alarm services to the scheduler. This module uses the type `task_struct`.
- The module `module.c` is not called from any modules in the kernel subsystem. It provides functions for registering, initializing, and freeing dynamically loadable modules. It also provides data structures for managing these modules. These functions would be called from outside the kernel subsystem.
- The module `panic.c` is not called from any modules in the kernel subsystem. It provides functions for initiating a kernel panic in the event of some fatal unrecoverable error. The panic call could come from any place in the kernel such as the memory management or file system subsystems.
- The module printk.c is called from the modules exit.c, dma.c, resource.c, `sched.c` and `softirq.c`. In all cases these modules are calling a function which prints information to stdout. This module makes use of the type `tty_struct`.
- The module `resource.c` is not called from any of the modules in the kernel subsystem. It provides functions which allow functions in other subsystems to request, reserve and free kernel resources (in particular io-regions).
- The module `signal.c` is not called from any modules in the kernel subsystem. It provides modules outside the kernel subsystem with functions for manipulating kernel signals.
- The module `softirq.c` is called only from the module `sched.c`. The call deals with "bottom-half" interrupt handling.
- The module `sys.c` is called from the modules exit.c, `sched.c`, `fork.c`, and `exec_domain.c`. The calls all deal with setting priorities and permissions of processes. This module uses the types `task_struct`, `inode`, and `file`. Note that Rigi was confused by a macro (`for_each_task()`) used by `sys.c`, but in fact, defined in `include/sched.h`. Rigi misinterpreted the macro as a function declared by `sys.c` and as a result showed dependencies on this macro as being dependencies on `sys.c` which is clearly incorrect.
- The module `sysctl.c` is called from the modules `module.c`, `fork.c`, and `exit.c`. These calls all involve the freeing of various kernel resources. This module uses the type `cff_table_header`.
- The module `time.c` is not called from any modules inside the kernel subsystem. It provides functions used by modules outside the kernel subsystem to get and set the time of day.

3.4 The Memory Manager Subsystem

Figure 6 shows the Memory Manager subsystem. The figure shows all the modules in the system and how they are connected by calls. The figure also shows the global types (structures) shared by the modules in the subsystem. In this case the arcs to these modules have been removed to improve readability. A list of types used by each module is given below.

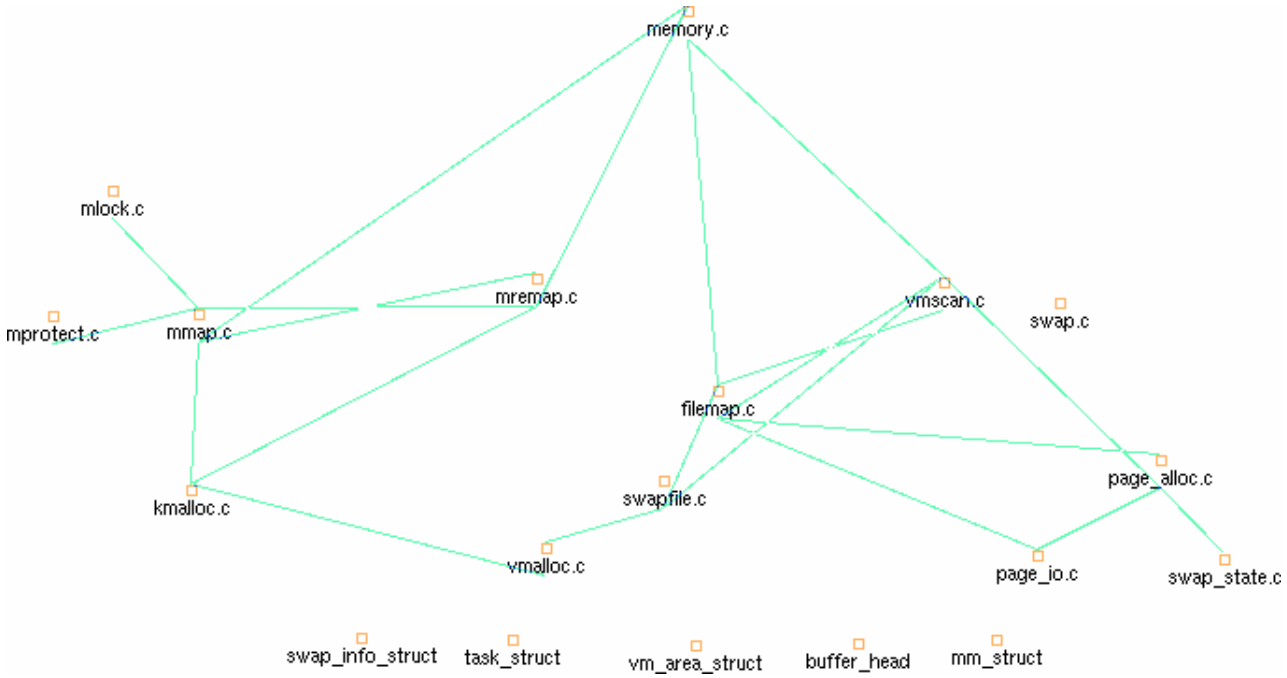

Figure 6: The architecture of the Memory Manager subsystem as extracted by Rigi.

- It is not clear what the primary module in this subsystem is. The `memory.c`, `mmap.c` and `filemap.c` modules appear to contain the core of the memory manager subsystem.
- The module `memory.c` is not called from anywhere. It makes use of the types: `vm_area_struct`, `task_struck` and `mm_struct`.
- The module `vmscan.c` is called by `memory.c`, `pagealloc.c`, `filemap.c`, and `swapfile.c`. These calls all deal with the freeing of dirty memory pages (after they are swapped out) and the management which goes along with this. This module makes use of the types `task_struct` and `vm_area_struct`.
- The module `mmap.c` is called from the modules `mlock.c`, `mprotect.c` and `mremap.c`. The first two modules make calls to manage the protection pages and the `mremap.c` module makes calls to perform page maps. This module makes use of the types `mm_struct` and `vm_area_struct`.
- The module `filemap.c` is called from `memory.c`, `vmscan.c` and `swapfile.c`. All these calls deal with the managing of memory mapped files. This module makes use of the types `vm_area_struct` and `buffer_head`.
- The module `page_alloc.c` is called from only `filemap.c`. These calls all deal with freeing pages which have been invalidated by `filemap.c`.
- The module `page_io.c` provides data to the modules `filemap.c` and `pagealloc.c`. This module makes use of the type `swap_info_struct`.
- The module `swap_state.c` is called from only `pagealloc.c`. The calls deal with initializing swap state. This module makes use of the type `swap_info_struct`.
- The module `swap.c` is not called by any modules in the memory management subsystem.
- The module `kmalloc.c` is called by `mmap.c`, `mremap.c`, and `vmalloc.c`. All calls deal with the allocation of memory.

- The module `vmalloc.c` is called by `swapfile.c`. The call deals with the management of swapspace.
- The module `mremap.c` is called only by `mmap.c`. The call deals with doing memory maps. This module makes use of the type `vm_area_struct`.
- The module `swapfile.c` is not called by any modules in the memory management subsystem. This module makes use of the types `swap_info_struct`, `task_struct`, `mm_struct`, and `vm_area_struct`.

3.5 The IPC Subsystem

Figure 7 shows the IPC (interprocess communication subsystem). The figure shows all the modules in the subsystem and how they are connected (by calls). The figure also shows the global types (structures) shared by these modules. In this case arcs indicate which modules use which types (types are all at the bottom of the figure). The three modules `msg.c`, `shm.c`, and `sem.c`, implement three types of interprocess communications mechanisms provided in Linux namely: message queues, shared memory, and semaphores. Pipes are also a method of interprocess communication but are implemented in the file system subsystem since they are more closely related to files (see `fs/pipe.c`).

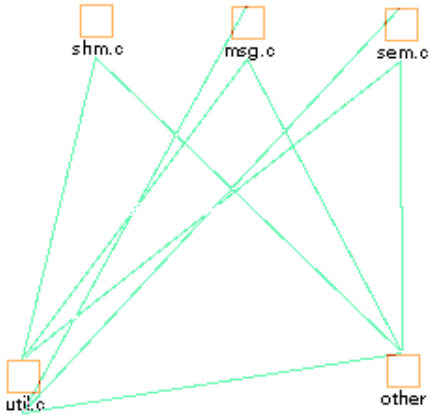

Figure 7: The architecture of the IPC subsystem as extracted by Rigi.

- The `msg.c` module provides general message serivces to users outside the system. It is used by the `util.c` module which calls `msg_init()` to initialize the `msg.c` module.
- The `shm.c` module implements the shared memory communications mechanism. It is called by the `util.c` module to perform initialization (`shm_init()`).
- The `sem.c` module implements semaphores services. It is initialized by the `util.c` module through a call to `sem_init()`.
- The `util.c` module is called by every module in the IPC subsystem. In each case the call is to the function `ipcperms()` which checks for permission to access ipc resources.
- The box labeled other in the figure represents dependencies on external modules as well as dependencies on data structures such as `shm_info`, `msg`,`sem_queue`, `sem_undo`, `vm_area_struct`, and `sem`.

3.6 The Network Subsystem

The network subsystem provides the high-level network support for the kernel. This does not include specific device drivers which can be found in the `drivers/net` subsystem. The structure within the network subsystem can be classified into two types: protocol independent modules and protocol dependent modules. The former category includes the `core` group, `socket.c`, and the `unix` group while the rest (i.e. 802, ipx, appletalk, ethernet, ax25, etc.) are in the latter category. Figure 8 shows the extracted modular decomposition for the Network subsystem. The box labeled external in Figure 8 represents dependencies on modules that are external to the network subsystem such as network drivers and memory management routines.

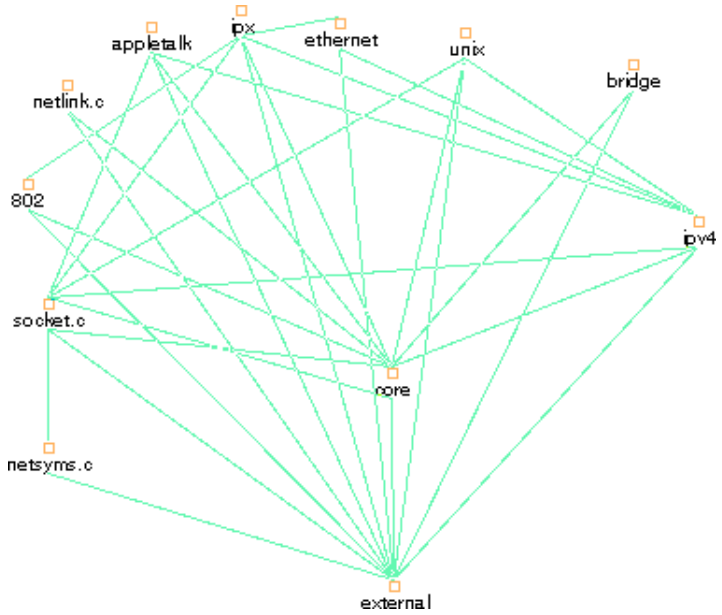

Figure 8: Network subsystem modular structure.

The `socket.c` module implements the top level API to the BSD socket library. The socket data structure holds information about BSD sockets and are associated with a file system inode. Sockets are thus treated as files but the socket.c module also defines special operations that can be done on sockets in addition to the standard file operations.

The `ethernet, 802, ipv4, ax25, ipx, appletalk,` and `bridge` groups contain protocol specific code for the respective network protocols. The `ipv4` group in particular contains a large body of code that implements the TCP/IP protocols.

The `core` group in the network subsystem, shown in Figure 9, contains modules that implement the core network functionality. This includes socket management routines and related data structures. The modules `sock.c` and `skbuff.c` contain most of the functionality related to the socket support routines. These have not been collapsed to show the relationship more clearly. The `sock.c` functions are to the right of the figure while the `skbuff.c` functions are to the left in solid colors. The `skbuff.c` module handles all manipulation of the Socket Buffer (`sk_buff`) data structures which are used to transfer data between protocol layers in a protocol independent manner (see [7] chapter 10.5). The `dev.c` module contains protocol independent device support routines which make use of socket buffers. The `datagram.c` module contains protocol independent datagram handling routines. In particular, the `select()` and `recvmsg()` code for protocols such as UDP, IPX, AX.25 are identical and thus use the datagram module.

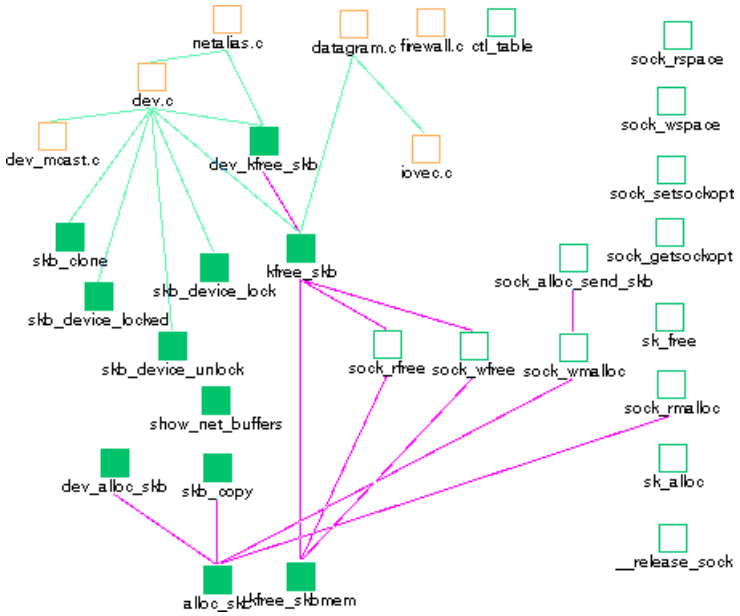

Figure 9: Network core subsystem modular structure.

3.7 The File System Subsystem

The file system can be conceptualized as two components: The Virtual File System (VFS) and the Specific File Systems (SFS).

The VFS describes all files in terms of superblocks and inodes. The inode data structure describe files and directories within the system; the contents and topology of the VFS. Each SFS, upon initialization, registers itself with the VFS. When a SFS is mounted the VFS must read its superblock by calling the SFS superblock read routine. Each SFS superblock read routine must map the specific file system topology into a VFS superblock data structure. Thereafter, the VFS routes file operations through the appropriate SFS. the VFS superblock manipulation routines are contained in the module `super.c`.

The file system makes use of three caches to help speed up file operations: the inode cache keeps recently accessed inodes, the directory cache contains information about recently accessed directories, and the buffer cache contains recently accessed raw file data from all the SFSes. The `dcache.c` module contains the directory cache implementation and the buffer cache code is contained in the `buffer.c` module.

As the system's processes access directories and files, system routines are called that traverse the VFS inodes in the system. System calls are contained in the following modules: `dquot.c, exec.c, fcntl.c, filesystems.c, locks.c, namei.c, open.c, read_write.c, select.c, stat.c, and super.c`. Most of these modules are shown at the top of Figure 10 and use the other modules in the file system subsystem to accomplish their tasks.

There are numerous modules in the file system subsystem. A number of these are described below.

- The `filesystems.c` module contains the initialization code for the subsystem which conditionally initializes each of the SFS that have been configured in the kernel.
- The `dquot.c` and `noquot.c` modules contain a real implementation and a non-implementation (transparent) of the diskquota system respectively.

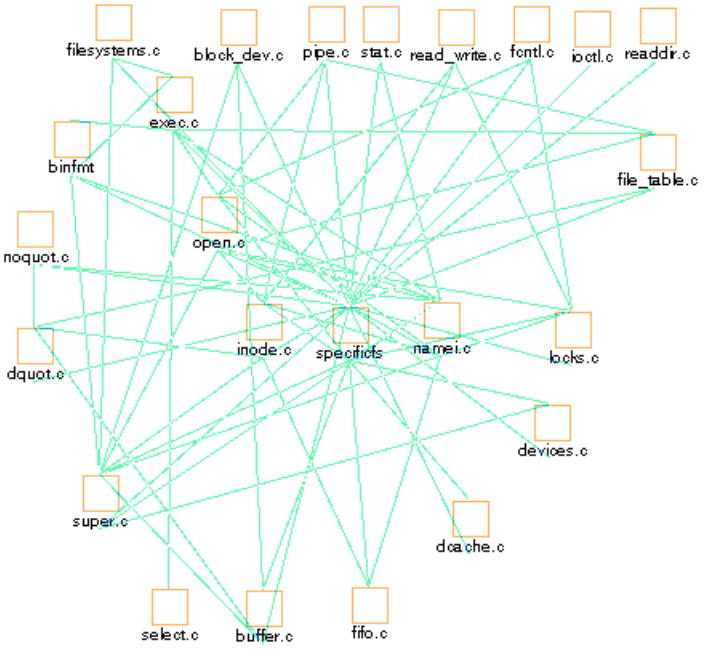

Figure 10: File subsystem modular structure.

- The `namei.c` module does filename and permission manipulation for the VFS.
- The `locks.c` module provides support for file locking and unlocking.
- The `devices.c` module implements the special operations supported for device files.
- The `open.c` module implements a large number of system calls such as `chmod()`, `chown()`, `open()`, `close()`, `chdir()`, `access()` etc.
- Linux supports several different binary formats such as AOUT, ELF, and even Java. These are implemented in their respective modules in the `binfmt` group.
- `read_write.c` implements the system call routines for reading and writing and seeking in files.

3.8 The Drivers Subsystem

The Drivers subsystem is the largest subsystem in the Linux kernel. It provides users of Linux with the ability to interface with a large number of hardware devices from hard disk drives and cd-rom drives to sound cards to network adapters. The Linux drivers are essentially shared low level hardware handling code that alleviates user applications from having to manage hardware controllers directly.

One of the basic features of UN*X is that it abstracts the handling of devices behind the file system in the form of *device special files*. Linux identifies devices by the device major number which is defined in `include/linux/major.h`. The three major types of devices that Linux supports are character devices, block device and network devices. Other types of supported device drivers are PCI bus, S-bus (SPARC), Sound drivers, ISDN drivers, and SCSI drivers. Figure 11 shows the structure of the drivers subsystem.

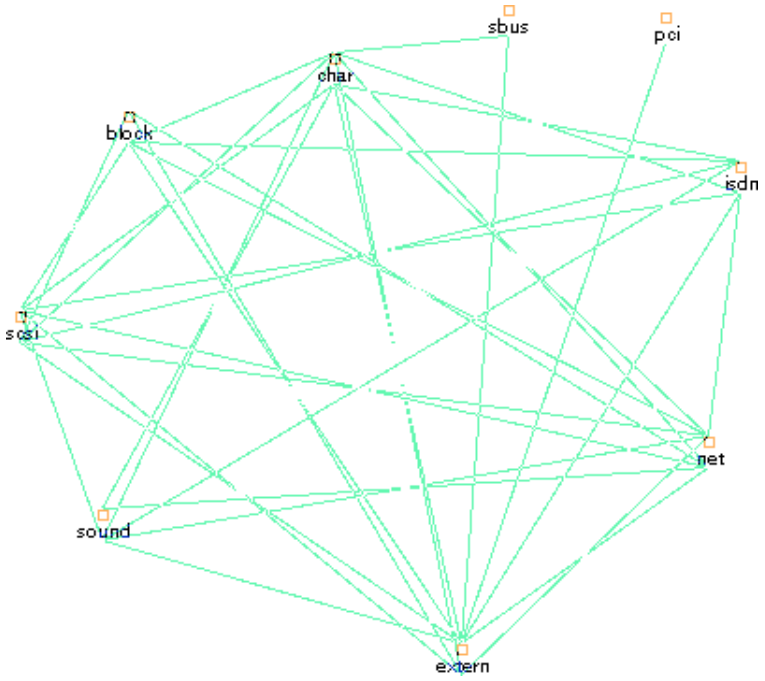

Figure 11: Drivers subsystem modular structure.

3.8.1 Block Devices

The block devices group contains drivers for block devices such as hard-disk, floppy, and cd-rom drives. Block devices can only be read and written to in multiples of the block size. Block devices are random access and make use of the buffer cache in the fs subsystem. Block device can be accessed via the device special files but are more commonly accessed indirectly via the file system. There are various specific drivers (`umc8572.c, qd6580.c, raid0.c, promise.c, ali14xx.c`, etc.) and a few more generic drivers (`hd.c, floppy.c, rd.c,` etc.). As can be seen in Figure 12 there is little coupling between modules. There are, however, dependencies by and on other subsystems which are not shown by this figure.

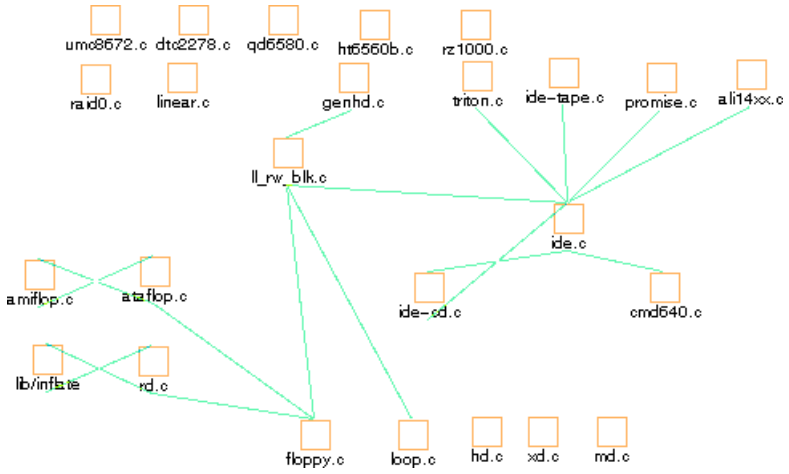

Figure 12: Block drivers modular structure.

- The `ide.c` module is the multiple IDE interface driver which handles devices that use the IDE interface.
- The `ll_rw_blk.c` module handles all read/write requests to block devices.
- The `rd.c` module implements a ramdisk driver and supports direct compression/decompression via gzip (inflate module in the lib subsystem).
- The `hd.c` module implements the hard-disk driver.
- The `xd.c` module contains the driver for an XT hard disk controller.
- The `md.c` module is a driver for multiple devices.
- The floppy disk driver is implemented by the modules `floppy.c`, `amiflop.c` and `ataflop.c`. These last two implement Amiga and Atari drives respectively.

3.8.2 Character Devices

Character devices are read and written to directly without buffering. There are numerous character device drivers including the serial ports, keyboard, mouse, and console screen among others. As can be seen in Figure 13 the character drivers are tightly coupled to the tty driver. The key modules are described below.

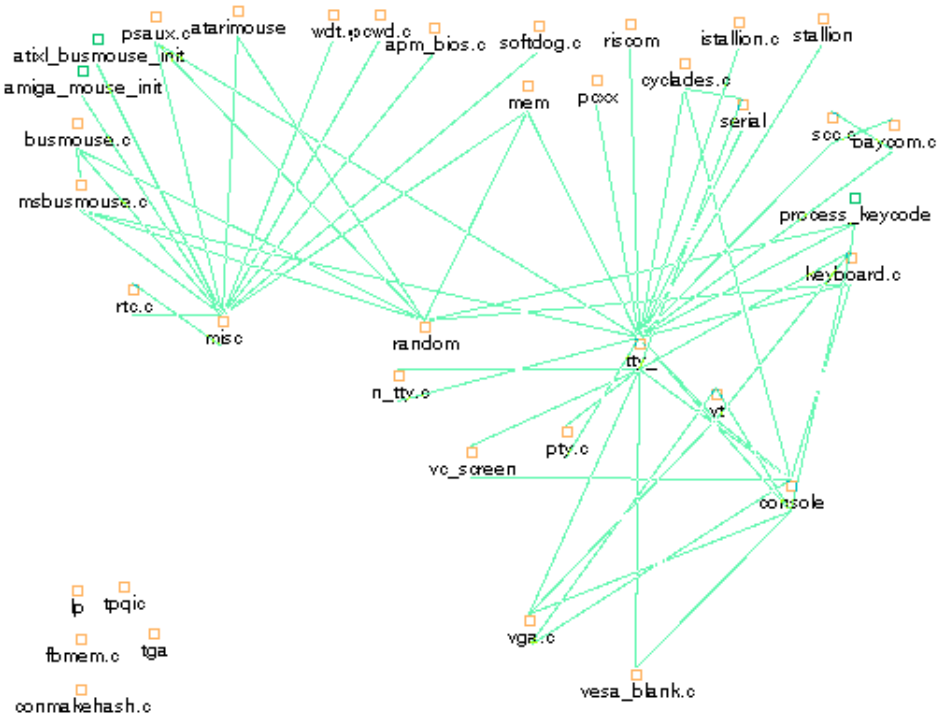

Figure 13: Character device drivers modular structure.

- ttys are a central component of this subsystem. The `tty_io.c` and `tty_ioctl.c` modules contain high-level tty routines while low level tty routines are contained in `serial.c`, `pty.c`, and `console.c`.

- `misc.c` contains generic misc open routines. (Whatever that means!) All the modules that use it call misc_register().
- Curiously the random number generator is located here (`random.c`) since it gathers environmental noise from device drivers to generate truly random numbers. It is also implemented as a character device and can be accessed via the device special file `/dev/random`.
- `stallion.c, istallion.c, riscom, and cyclades.c` are multiport serial driver modules. `serial.c` is the generic Linux serial driver.
- `keyboard.c` implements the keyboard driver.

3.8.3 Network Devices

This subsystem contains various hardware specific network card drivers. As well as a few software ones such as the loopback adapter (`loopback.c`) the serial line protocol (`slip.c`) and the point to point protocol (`ppp.c`).

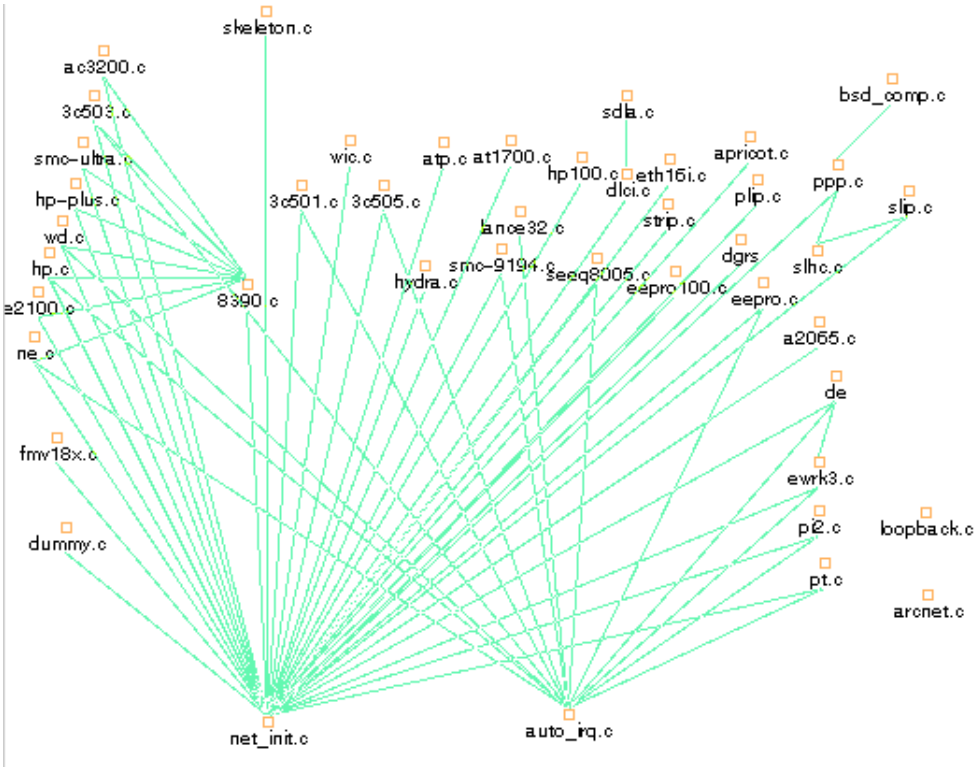

Figure 14: Network drivers modular structure.

- The `net_init.c` module contains initialization routines for "pl14+" style ethernet drivers.
- The `8390.c` module contains the chip specific code for many 8390-based ethernet adaptors. It must be combined with board-specific code to make it a complete driver but it implements the core functionality for these types of boards.
- The `auto_irq.c` module is a general-purpose IRQ line detector for devices with jumpered IRQ lines. This code allows these drivers to detect which IRQ line the hardware is using.

- The module `skeleton.c` is a template which can be used as a guide for writing network device drivers.
- The `slhc.c` modules contains routines to compress and uncompress tcp packets (for transmission over low speed serial lines). Hence the dependency of ppp and slip on this module.

4 Conclusion

In this report we have provided a detailed examination of the Linux Kernel architecture. This concrete architecture was put into the context of our conceptual (or abstract) architecture which was the subject of previous reports [3, 4, 6]. A high level overview of the concrete architecture was given and this was followed by a detailed examination of the architecture of each subsystem in the Linux kernel.

Architecture descriptions were supported by figures generated using the Rigi system. This work provides an indication of how this system can be used in similar reverse engineering or extraction project. The report also comments on some of the limitations of the Rigi system and some of the anomalies which we encountered.

This report achieves its goal of documenting the concrete architecture of the Linux kernel. Performing this extraction proved to be both challenging and interesting and certainly provided the authors with practical experience in reverse engineering and architecture extraction.

A Glossary

`datagram.c` This module in the network subsystem provides protocol independent datagram handling routines.

Device Drivers Software which provides the interface between higher levels of the operating system and the hardware devices such as disks and network cards. Usually very complex code.

`dev.c` This module of the network subsystem provides protocol independent device support for routine which use sockets.

`dma.c` This module within the kernel subsystem deals with direct memory access.

`exit.c` This module within the kernel subsystem deals with killing, resetting and otherwise exiting processes. It also deal with the reset of the entire OS.

`filemap.c` This module in the memory management subsystem manages memory mapped files.

`filesystem.c` This module in the file system subsystem performs basic file system initialization.

`fork.c` This module within the kernel subsystem manages the forking of processes.

`ide.c` This module in the device driver subsystem provides interface support for IDE devices.

`ide_setup` A procedure call made from `main.c` in the Init subsystem. One of the many initialization calls made. This one deals with setting up the IDE device driver.

`itimer.c` This module with the kernel subsystem contains modules for keeping track of various timers and sounding alarms when they expire.

Inter Process Communication (IPC) Communication between two processes through pipes, sockets, signals, or shared memory (and others).

Linux A UNIX like operating system written from scratch by Linus Torvalds.

`main.c` This is the bootstrapping module in the Init subsystem. This module is responsible for initialization and startup.

Memory Manager The part of the kernel responsible for managing virtual memory. Includes paging, swapping and memory mapping .

`mmap.c` This module in the memory management subsystem is responsible for managing all memory mapping operations.

`mm_struct` A commonly used structure of the memory management subsystem.

`msg.c` This module in the IPC subsystem provides generic message handling routines.

`panic()` This function is provided with the `panic.c` module of the Kernel subsystem. It can be called from nearly everywhere in the Linux kernel. It is normally called when unrecoverable errors occur. It allows modules in other subsystems to initiate a kernel panic.

Network Services Manager The part of the kernel responsible for managing all interactions with a network.

Pages In terms of memory management, a page is a logical part of virtual memory which can be loaded into physical memory or written to disk.

`page_alloc.c` This module in the memory management subsystem deals with the allocation and deallocation of memory pages.

Process Manager The part of the kernel responsible for managing processes. This is where time slicing is implemented. The process manager uses a scheduler to determine which process runs next.

`sched.c` This module in the kernel subsystem is responsible for scheduling processes within the kernel.

`sem.c` This module in the IPC subsystem provides support for semaphore use in interprocess communications.

Specific File System This refers to a particular type of file system such as msdos, minix, ext2, ext, or iso9660 file systems.

`signal.c` This module in the kernel subsystem is responsible for managing signals.

`socket.c` This module of the network subsystem provides an API to the BSD socket library.

`super.c` This module in the file system subsystem provides routines for manipulating VFS superblocks.

`softirq.c` This module in the kernel subsystem is responsible for handling interrupt requests.

`swap.c` This module in the memory management subsystem provides service routines used in memory swapping.

`task struct` A common data structure used in the kernel subsystem. It contains data regarding process tasks.

Time Slicing An method of sharing a CPU among more than one process. A process may *have* the CPU for a fixed amount of time but then must give it up for another process to be scheduled to run.

Virtual File System In Linux, the file system is broken into two parts. One part contains the file system non-specific portions, this is the virtual file system. The other part is the specific files system (SFS). this way Linux can support many different kinds of file systems simultaneously.

`vm_scan.c` This module of the memory management subsystem is responsible for freeing dirty memory pages after they have been swapped out.

`vm_area_struct` A common data structure used in the memory management subsystem. Used to keep track of virtual memory.

`wait_queue` A common data structure in the kernel subsystem. Used by the scheduler to keep track of processes waiting to be run.

B Abbreviations

API Applications Programming Interface

CPU Central Processing Unit.

GUI Graphical User Interface.

IPC Interprocess communication.

ISDN Integrated Systems Digital Network.

MM Memory Manager.

NFS The Networked File System.

SCSI Small Computers System Interface.

SFS Specific File System.

References

[1] Linux Documentation Project at: http://sunsite.unc.edu/LDP/linux.html#ldp.

[2] Johnson, M., **The Linux Kernel Hacker's Guide**, at: http://www.redhat.com:8080/HyperNews/get/khg.html.

[3] Kamel, M., "Abstract Architecture of the Linux Kernel", January, 1998. http://swen.uwaterloo.ca:80/~m2kamel/research/a1.ps.

[4] Keast, J., "The Linux Kernel: A Conceptual Architecture", January 1998. http://keiths.uwaterloo.ca/linux.

[5] *Linux Source Code 2.0.30.* Available at: http://www.kernel.org.

[6] Pal, C., "The Abstract Architecture of the Linux Kernel", January 1998. http://ozone.crle.uoguelph.ca/chris/linux/abstract.html.

[7] Rusling, D., **The Linux Kernel**, DRAFT version 0.1-13(19), July 1997. available at: ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/tlk-0.1-13-19.ps.gz.

[8] Shaw, M., Clements, P., "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", C.S. Dept and Software Engineering Institute, Carnegie Mellon Univ, Pittsburgh, PA 15213, 1996.

[9] Tanenbaum, A., S., **Modern Operating Systems**, Prentice Hall, 1992.

[10] *The Rigi Project.* http://tara.uvic.ca.

[11] Tama Communications, "Linux Kernel Tour" at: http://wafu.netgate.net/linux/index.html.