Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Linux in Safety-Critical Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Roland Kammerer
Matrikelnummer 0125555

an der
Fakultät für Informatik der Technischen Universität Wien
ausgeführt am
ICT - Institut für Computertechnik

Betreuung:
Betreuer: O. Univ. Prof. Dipl.-Ing. Dr. techn. Dietmar Dietrich
Betreuer: Dipl.-Ing. Andreas Gerstinger

Wien, 04. 11. 2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Kurzfassung

Die heutige Gesellschaft ist von einer Vielzahl von sicherheitskritischen Systemen abhängig, die die Sicherheit der Benutzer und der Umwelt garantieren müssen. Deshalb ist es entscheidend welches Betriebssystem auf solchen sicherheitskritischen Systemen zum Einsatz kommt. Diese Arbeit untersucht einen möglichen Einsatz des Kernels Linux und des Betriebssystems GNU/Linux für sicherheitskritische Systeme. Um Vertrauen zu schaffen, dass GNU/Linux als Plattform für sicherheitskritische Anwendungen eingesetzt werden kann, wurde anhand von relevanter Literatur erforscht, wie Linux entwickelt und getestet wird. Das Open Source Entwicklungsmodell wurde mit traditionellen Entwicklungsmodellen vergleichen. Es wurde analysiert, ob der aktuelle Stand der Entwicklung die Anforderungen von existierenden Normen im Bereich sicherheitskritischer Anwendungen erfüllen kann. Um die Relevanz von Linux für sicherheitskritische Anwendungen noch weiter zu erhöhen, wurden zwei Projekte implementiert. Das erste ist eine vollautomatische Testumgebung für RAID-1 Systeme auf Kernelebene, das zweite ist ein Dateisystem, das schadhafte Daten erkennt und korrigiert. Es zeigt sich, dass das Verfahren der Open Source Entwicklung, das von der GNU/Linux Entwicklergemeinschaft eingesetzt wird, kein limitierender Faktor ist. Die meisten Standards, die sich mit sicherheitskritischen Systemen beschäftigen, sind flexibel genug, um Systeme, die auf neuen und offenen Wegen entwickelt wurden, zu zertifizieren. Dies führt zum Ergebnis, dass Linux als Plattform für sichheitskritische Anwendungen in Betracht gezogen werden sollte. Die Entwicklung von Linux hat in den letzen Jahren einen großen Fortschritt gemacht, der Linux reif für den Einsatz in sicherheitskritischen Systemen macht.

Abstract

Modern society depends on a range of systems that need to guarantee the safety of their users and the environment. Therefore it is crucial which operating system is used for such safety-critical systems. This thesis examines the potential use of the operating system kernel Linux, and the GNU/Linux operating system for safety-critical systems. To gain confidence that GNU/Linux can be used as a platform for safety-critical applications, it was examined how Linux is developed and tested by assembling information from relevant literature. This Open Source development model was compared to traditional software development models. It was analyzed, if the current state of development can fulfil the requirements of existing safety-related standards. To further improve the relevance of Linux for safety-critical systems, two projects have been implemented. The first one is a fully automatic test suite for kernel-level software RAID-1 systems, the second one is a wrapper file system that detects and corrects faulty data on hard disks. It turns out that the Open Source development approach taken by the GNU/Linux community is not a limiting factor for its use in safety-critical applications. Most standards that deal with safety-critical systems are flexible enough to certify systems that are developed in new and open ways. As a result, Linux should be considerd as a platform for safety-critical systems. The development of Linux made large progress during the last years, which makes Linux fit for safety-critical systems.

Acknowledgements

First of all I would like to thank my parents for giving me financial and personal support during my whole lifetime.

Thanks to Andreas Gerstinger for his endless patience and the ability to talk to people involved in Linux kernel development. Thanks to Wolfgang Puffitsch for reviewing my source code and to Daniel Schneider for proof-reading. Thanks to my fellow students and true friends Harald Krapfenbauer and Wolfgang Puffitsch for all their support and interesting discussions during the last years.

Last but not least I would like to thank all the Linux kernel hackers and the GNU project for providing such great software, which makes computer science that much fun for me.

Table of Contents

Abbreviations

API	Application Programming Interface
BSD	Berkeley Software Distribution
COTS	Commercial/Component Off The Shelf
CPU	Central Processing Unit
CVS	Concurrent Versions System
FC	Failure Case
FMEA	Failure Mode and Effect Analysis
FOSS	Free and Open Source Software
FS	Free Software
FTA	Fault Tree Analysis
GDB	GNU Project Debugger
GNU	GNU's Not Unix
GPG	GNU Privacy Guard
GPL	GNU Public Licence
GUI	Graphical User Interface
HRT	Hard Real-Time
HTML	HyperText Markup Language
IPC	Inter Process Communication
ISV	Independent Software Vendor
KVM	Kernel-based Virtual Machine
LF	Linux Foundation
LSB	Linux Standard Base
LTP	Linux Test Project
MIT	Massachusetts Institute of Technology
MMU	Memory Management Unit
MTA	Mail Transfer Agent
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure

MTTR Mean Time To Repair
MULTICS Multiplexed Information and Computation System.
NC Normal Case
NIC Network Interface Card
NPTL Native POSIX Thread Library
NUMA Non Uniform Memory Access
OEM Original Equipment Manufacturer
OSADL Open Source Automation Development Lab
OSDL Open Source Development Labs
OSS Open Source Software
PGP Pretty Good Privacy
POSIX Portable Operating System Interface
QoS Quality of Service
RAID Redundant Array of Inexpensive Disks
RAMS Reliability, Availability, Maintainability, and Safety
RHMRG Red Hat Messaging, Real Time Grid
RTDM Real-Time Driver Model
SCI System Call Interface
SCSI Small Computer Systems Interface
SC Special Case
SHA Secure Hash Algorithm
SIL Safety Integrity Level
SLES SUSE Linux Enterprise Server
SMP Symmetric Multiprocessing
SOUP Software Of Unknown Pedigree
SSL Secure Socket Layer
SVN Subversion
TKO Test Kernel Org
TMR Triple Modular Redundancy
TTA Time Triggered Architecture
TTP Time Triggered Protocol
UML User Mode Linux
USB Universal Serial Bus
VFS Virtual File System

1 Introduction

The introduction gives a short overview about the problem, the intentions of this thesis and the methods that are used.

1.1 Problem Description

In the modern world mankind depends on quite a number of safety-critical systems. Sometimes this dependability is obvious as on the subject of aircraft construction and sometimes it is hidden in tiny embedded systems like ones that trigger a fire alarm. The more sophisticated a system is, the more likely these systems need some kind of software which controls their functions.

One of the most fundamental software components in such a safety-critical system is the operating system which is the base for all higher level applications. Probably the most important part of the operating system is a piece of software which is called the *kernel*. The kernel manages the access to the underlying hardware. This thesis concentrates on a specific kernel named *Linux* [27], which is the kernel of the so called GNU/Linux operating system [15]. It is essential to choose an operating system, and therefore a kernel, which guarantees a high degree of Quality of Service (QoS). There are lots of important attributes like *reliability*, *availability*, *maintainability*, and the one which is most important in the context of this thesis, the *safety*. Together these properties are called "RAMS". These terms will be discussed in detail in section 3.2. A more generic name which includes the above mentioned terms and extends them by properties like *security* is called *dependability*.

One of the most important questions to ask is how can safety be measured and how is it possible to certify an operating system according to these levels of safety? This question will lead to so called *Safety Integrity Levels* (SIL) which can be used to distinguish different levels of safety requirements. The SIL gives information about the criticality of a system. On the basis of these requirements someone needs to certify the system according to certain rules, the *safety standards*. With this certification it is possible to guarantee safety to users and the environment.

As it can be imagined, safety-critical systems are a complex matter and therefore it can be an advantage if software (especially operating systems) can be reused. In general there are several factors which recommend software reuse. It is possible to reduce development costs, it may be faster to use an already existing product than developing a new one and last but not least, it is possible to buy in expert know-how from specialized offerers. This strategy can be summarized under the term *Commercial/Component Off The Shelf* (COTS) or *Software Of Unknown Pedigree* (SOUP).

1.2 Intentions

The intention of this thesis is to discuss the possibility of using *Linux*, the kernel, for safety-related applications. The kernel is one major part of the operating system and therefore it is valuable to concentrate on that part, but on the other hand an operating system is much more than a kernel, therefore this thesis will focus on aspects of the whole GNU/Linux operating system whenever it is important for the completeness of this thesis. The focus is on the domain of railway systems, which has several special requirements that differ from traditional domains like aircraft construction. These differences will be discussed in the chapters that concentrate on safety.

To prove that GNU/Linux can be used for safety-critical systems, different topics have to be examined. This includes the software development model the Linux community uses, which gets compared to more traditional ones. A review, if this development model fulfils the requirements for safety-critical systems according to well known standards, is given.

The question if it is advisable to use COTS software for this form of application domain will be discussed in section 3.5. Different COTS systems are examined and classified by their use in safety-critical system.

A practical part (section 5) examines the extendibility of GNU/Linux and the methods of testing existing components. A suite for testing RAID-1 systems will be presented and additionally the implementation of a wrapper file system that provides safety-related features will be shown.

The goal of this thesis is to provide information about the possibility of using Linux, or modified versions of Linux, for SIL4 (the most critical safety integrity level) applications.

1.3 Method

The work is mainly based on the study of relevant literature. Important documents for the discussion of relevant work in this context are the research report "Preliminary Assessment of Linux for Safety Related Systems" [Pie02], written by Ron Pierce, and the diploma thesis "COTS components in safety-critical systems" [Kri02] by Linda Kristiansen. To improve the trust in Linux and GNU/Linux components, a practical part is shown, that follows a hands-on imperative and presents two examples of the extendibility and testability of Linux and the GNU/Linux operating system.

2 Linux and GNU/Linux

The kernel is one of the most important parts of an operating system. The kernel manages the access to the hardware and therefore it is a layer, which is between the hardware and the so called user space programs. Without this piece of software an operating system cannot provide any useful services. It is essential to distinguish between the kernel and the operating system itself. An operating system consists of much more programs than only the kernel. For example an operating system provides compilers, editors, graphical user interfaces and many more services like ftp or web-server. Figure 2.1 gives an overview of the different operating system layers.

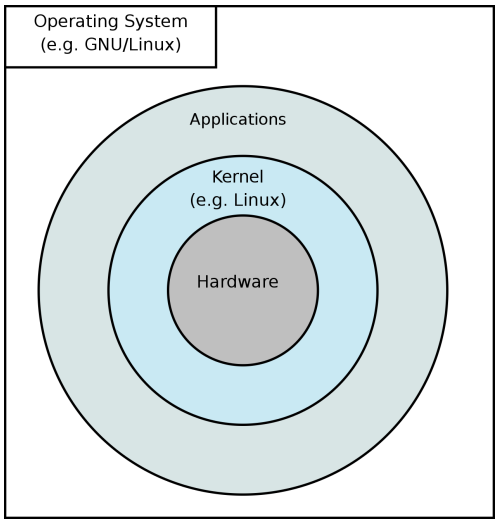

Figure 2.1: Structure of an operating system

If the term *Linux* is used it refers to the kernel itself. If the whole operating system is meant it is entitled as *GNU*[1]*/Linux*.

Before starting with a little history about GNU/Linux and UNIX in general, it is important to understand the concept of *Open Source Software* (OSS) and *Free Software* (FS). From a technical point of view these two terms are very similar, but there are philosophical differences, which will be examined. There is a special term which combines Free and Open Source Software, which is *FOSS* (Free and Open Source Software).

[1]The GNU project was announced in 1983 with the goal to create a free UNIX-like operating system.

2.1 Free and Open Source Software

The term *free software* was defined by the GNU project [15] to give the users of software several rights. Open Source software is mostly known for the right that everybody has access to the source code and that everybody has the right to alter it. This is one important part, form a pragmatic standpoint the most important one, but the concept of free software has an *ethical* standpoint, too. There is one famous statement that is cited over and over again which says: "Free software is a matter of liberty, not price. To understand the concept, you should think of *free* as in *free speech*, not as in *free beer*" [15]. The GNU project and the *Free Software Foundation* give the following definition of free software:

- Freedom 0: The freedom to run the program, for any purpose.
- Freedom 1: The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.
- Freedom 2: The freedom to redistribute copies so you can help your neighbour.
- Freedom 3: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

A program that includes all of the mentioned rights is *free software* as defined by the GNU project. There are several software licences that give the user these rights, the most common used is the GPL (GNU Public Licence). A list of licences that are compatible to the GPL can be found on the web page [19] of the GNU project.

On the other hand, there exists the term *Open Source* which is defined by the Open Source initiative [34]. The Open Source definition mentions the following rights:

- Free redistribution: The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
- Source code: The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.
- Derived works: The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
- Integrity of the author's source code: The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

- No discrimination against persons or groups: The license must not discriminate against any person or group of persons.
- No discrimination against fields of endeavour: The license must not restrict anyone from making use of the program in a specific field of endeavour. For example, it may not restrict the program from being used in a business, or from being used for genetic research.
- Distribution of license: The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
- License must not be specific to a product: The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.
- License must not restrict other software: The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.
- License must be technology-neutral: No provision of the license may be predicated on any individual technology or style of interface.

The conclusion from both definitions is that they are similar, but the term Open Source was created for marketing reasons because the term *free* sometimes confuses management people who associate *free* with "cheap" or "worthless". In general people interested in the ethical and philosophical standpoint use the term *free software*, whereas people who are more pragmatic, or when it comes to marketing, use *Open Source Software*.

2.2 History

Linux is derived from an operating system called UNIX and therefore the history is started with the roots of UNIX. UNIX started in 1969 as a project of the Bell Labs, which was a research facility of AT&T and Western Electric. Ken Thompsen, an employee of AT&T worked on an operating system called MULTICS in cooperation with General Electric and the MIT.

In March of 1969 Bell Laboratories quit the project, but Ken Thompsen decided to create an operating system which should provide the following features [Wie98]:

- Hierarchic file system to divide the memory in a flexible way.
- The output to a file and the output to a peripheral device should be similar.
- The possibility to start new processes independently from other processes.
- Every user can choose his favourite command interpreter.
- A rich set of utilities especially compilers for different programming languages.

- The operating system should be portable to new hardware architectures.

The following text summarizes Linux's history according to Wielsch [Wie98]: In 1972 Dennis Ritchie, Rudd Canaday, and Brian Kernighan, who gave the operating system the name UNIX presented a first prototype. To fulfil the last goal on the list, the portability, most of the UNIX kernel was written in a high level programming language named *C*. Until 1975 the UNIX operating system was developed exclusively by the Bell Laboratories. With version 7, the operating system had its first commercial success and it got ported to several hardware platforms by companies like Data General, PCS, Digital Equipment, Amdhal, and Hewlett Packard. Important improvements were made by the University of California, which named their operating system BSD (Berkeley Software Distribution). This was necessary because the name UNIX was trademarked.

In 1987 Professor Andrew S. Tanenbaum, who was working at the Vrije University in Amsterdam released a UNIX like operating system called *Minix*. He wanted to create a simplified UNIX system to be able to show his students how operating systems work. For non commercial use Minix was delivered free of charge.

In 1991 a Finnish computer science student named Linus Torvalds decided to program his own operating system kernel inspired by the ideas of Minix. The motivation of his project was to replace the Minix kernel with his own kernel, Linux. Therefore Linux was bound to the Minix userland in its early days. The kernel hackers adapted Linux to operate with the GNU components. The GNU project itself was founded in 1983 with the goal to create a free UNIX-like operating system. In the 1990s the GNU project had developed a bunch of sophisticated software like compilers, editors, and libraries but the heart of its operating system, the kernel, was missing. This gap was filled by Linux. The resulting operating system is called *GNU/Linux*, the GNU system with Linux as one of its kernels. In March 1994 the first official version (1.0) of Linux was released. This release supported only the i386 single processor hardware with which Linus Torvalds was familiar. One year later version 1.2 had support for different hardware architectures like Sparc, Mips, or Alpha but it still was a kernel for single processors systems. This was changed with Linux version 2.0 in 1996 which introduced *Symmetric Multi Processing* support (SMP). Linux version 2.4 was an important step for Linux to the area of desktop computing. For example it had better support for so called Plug-and-Play hardware and USB devices. In 2003 Linux reached version 2.6, which is still the relevant development series nowadays. Its features will be discussed in section 2.3.

2.3 Features and Design

The kernel Linux offers interesting features that are presented afterwards. First in general and then for the current Linux version 2.6.

In general Linux is a full featured operating system kernel which provides features like multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, memory management, and multistack networking. One of its biggest advantages is its enormous scalability and portability. The kernel can scale form embedded devices to cluster computer arrays with thousands of nodes. According to to the kernel documentation Linux was ported at least to the following hardware architectures: 32-bit x86-based PCs (386 or higher), the Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH,

IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS, and Renesas M32R architectures[2].

Linux is designed as a *monolithic kernel* which means that all the services are integrated in the kernel itself. It is appropriate to think of Linux as one single binary file. This is in contrast to so called *micro kernels* that provide only very basic functions in the kernel itself whereas other services are located in different layers of the micro kernel architecture. There are several micro kernel approaches like *GNU/Hurd* [13] or the much more successful ones like *Mach, Amoeba and Chorus* [CDK94]. At the time of programming the first Linux versions, Linus Torvalds decided that a micro kernel does not fit his needs [TD02]. The architecture of the kernel was described by Tim Jones [2], which gave the base for the following summary. Torvalds monolithic approach led to the following architecture of the GNU/Linux system, which is shown in figure 2.2.

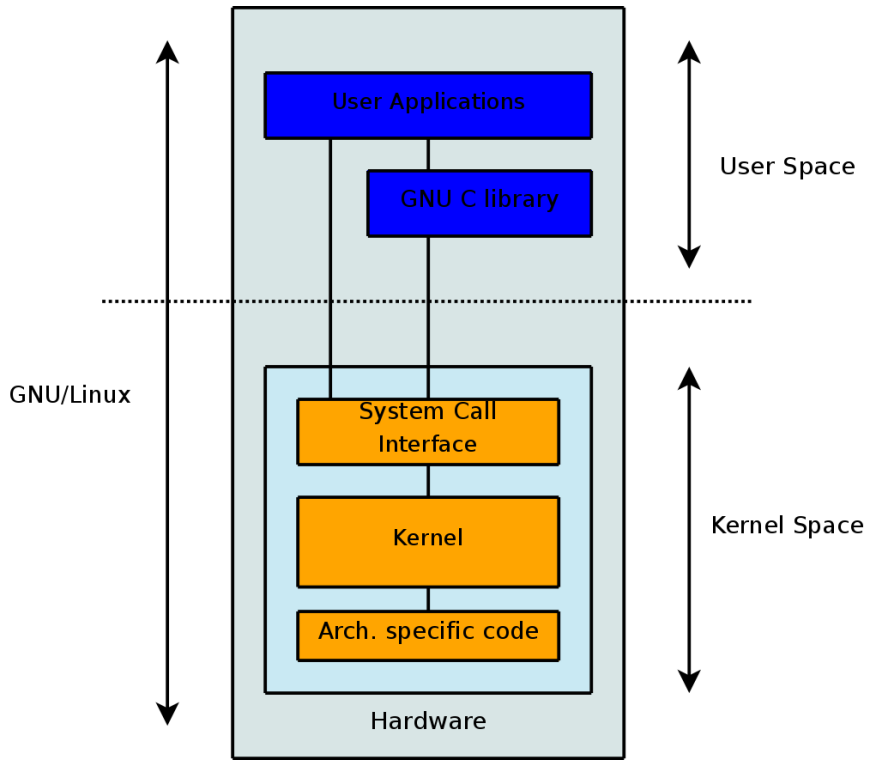

Figure 2.2: Basic GNU/Linux architecture

On the most abstract level there is a separation between kernel and user space. In user space each program has its virtual address space. This is the part of the kernel where all the programs are executed that an ordinary user wants to start. For example a web browser, an editor, or a desktop manager. On the other hand there is the kernel space with its own single address space. There is a layer that connects user space with kernel space. This layer is implemented with the *GNU C Library* (glibc) and the *System Call Interface* (SCI). Therefore in general a user space application is not allowed to communicate with the hardware itself, users send their requests to the SCI and get back an answer from the kernel. The kernel itself can be further divided in an architecture independent part and an architecture dependent part. The independent part

[2] /usr/src/linux/README

is shared between all the architectures Linux supports, whereas the dependent part deals with special hardware requirements for the specific hardware platform.

The kernel itself is divided into several major subsections that are shown in figure 2.3.

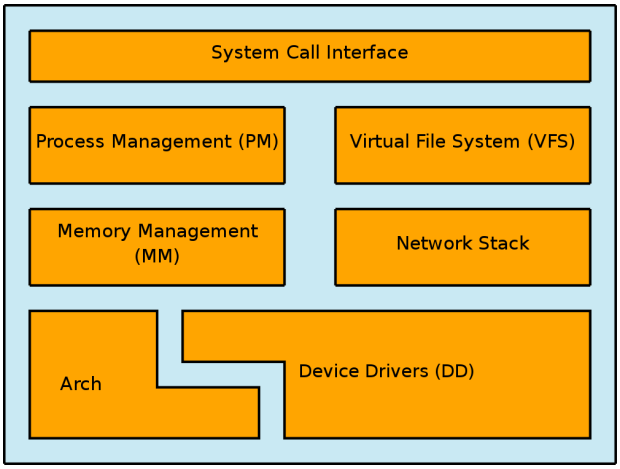

Figure 2.3: Basic Linux components

- Process management: This is the part of the kernel where the user space processes are managed. In the usual terminology there is a difference between *processes* and *threads*. Linux does not difference them and has the internal concept of *threads*. The kernel provides an interface through the system call interface where processes can be created, forked, executed, stopped, killed, and it manages the communication and synchronisation between processes. *Scheduling* is done in this part of the kernel, too, which means the strategy how many CPU cycles each process gets and when does it get them.

- Memory management: This is the part where the memory and virtual memory is managed by Linux. In most cases memory is managed in so called *pages* with a traditional size of 4 KB. Linux provides the ability to map virtual memory to physical memory and it is the place where *swapping* is done. Swapping needs do be done if all the memory is consumed and there is no memory left. Then Linux swaps out regions of memory to the hard disk. GNU/Linux has an own partition type "swap" for these kinds of partitions, but swapping can also be done to ordinary files on a file system. Allocating 4 KB blocks is not the only thing the kernel does in its memory management part. For example, it keeps track of the usage of memory and knows which pages are full, empty, or partial used.

- Virtual file system: The virtual file system is an abstraction between the user space and the different under-laying file system types. For example the user space application does not need to know if the under-laying file system is *ext2*, *ext3*, *vfat*, or any other of the supported file systems. The user space application sends the command "write to disk", and the kernel decides how this is done.

- Network stack: The network stack is built up in layers itself, which is known from the ISO-OSI model. For example there is an IP layer and above it is the TCP layer. The topmost layer are *sockets*. Sockets provide an user interface from the user space to the

kernel networking stacks. Of course Linux is not limited to TCP/IP, the kernel provides a large spectrum of networking protocols.

Joseph Pranevich [56] and Paul Venezia [31] give a detailed view of the features of Linux 2.6. There perceptions are summarized in the rest of this chapter.

One of the most important aims of Linux is the compliance to the POSIX [POS04] specification. In short, POSIX describes an application interface between the application and the corresponding operating system. POSIX is an *international standard* and therefore an operating system can be certificated according to it. One example is the *Native POSIX Thread Library* (NPTL) which runs on top of the rewritten threading infrastructure of Linux 2.6

The possibility of scaling down to embedded devices like PDAs or network-routers was achieved by integrating code from the *uCLinux* [54] project in the main kernel. The Linux microcontroller project has the goal to port Linux to devices that do not have a *Memory Management Unit* (MMU). These MMUs protect the memory from getting overwritten by other processes. The memory is divided in several sections and processes are allowed to write to their own sections but not to sections from other programs. The most critical event is when a process overwrites the memory of the kernel which leads to a crash of the system.

The other direction, scaling up, was achieved by better SMP and NUMA (*Non Uniform Memory Access*) support. Both concepts deal with computer architectures that have multiple CPUs (*Central Processing Unit*). While SMP systems share one big memory space, every CPU in a NUMA system has its own memory and even I/O components. The further step to NUMA was necessary because a single memory became the bottle neck for large server systems. History has shown that Linux 2.6 is a kernel that provides excellent support for these kinds of systems. Technically this was achieved by an internal "topology API", that represents the inner structure of each CPU and its components. Therefore the scheduler is able to exploit the locality of the memory and I/O components of each CPU in the NUMA system. Another concept that is related to this issue is *hyperthreading*, which is used by Intel's Pentium 4 architecture and others. Hyperthreading is a way to simulate multiple CPUs on only one physical CPU and it is done in hardware. While bringing speed enhancements this feature introduced complexity for the kernel's scheduler. Support for hyperthreading was introduced during Linux 2.6. In general Linux 2.6 scales much better than Linux 2.4 because limits got removed or at least extended to fulfil requirements of modern systems. For example the number of Process IDs (PID) was increased from 32000 to 1 billion or the number of individual users and groups was raised from 65000 to 4 billion. With Linux 2.4 it was possible to have 255 major devices with 255 minor devices (e.g. partitions). Linux 2.6 increased this numbers to 4096 major devices with a minor device number of more than one million devices.

Linux has its roots on the i386 platform and therefore CPU and architecture were coupled in a tight way. In the modern computer architecture world this assumption does not hold any more and therefore the concept of *subarchitectures* was introduced which split up the whole architecture in individual parts. This made Linux 2.6 much more portable to non-standard architectures. The concept of subarchitectures splits up the system in individual components wherever it is possible and wherever it makes sense.

The kernel itself is designed in a very modular way and therefore it is possible to integrate kernel support for exactly that type of hardware that is actually integrated in a device. If someone compiles a kernel from source code he or she has the chance to exclude unnecessary device

drivers. This reduces the resulting binary size which can be interesting for small embedded devices where size matters. This feature is attractive for developers of safety-critical systems, too. If a driver (for an unused hardware device) is not integrated in the kernel binary it cannot fail. With that modular kernel it is possible to reduce points of failure in an uncomplicated way. The standard kernel provides loading and unloading of so called *kernel modules*. These modules are object files, with the extension "*.ko" (kernel object), which can be loaded and unloaded at runtime. For example, it is possible to compile SCSI and USB mass storage support as modules. The resulting kernel image does not provide support for these kinds of hardware devices but if it is necessary someone can load them via the program `modprobe`. The module gets linked to the kernel at runtime and therefore the device can be used the same way as if the support would have been integrated in a static way. If unloading of kernel modules is considered as unsafe it can be disabled at all. In modern environments modules need to get information about the status of other modules. For example consider a module which is responsible for the power management. It needs information from the disc device drivers, or the scheduler needs information from the power management module. Modern architectures force interconnection and information exchange between subarchitectures. Getting this information was much more limited in Linux 2.4 than in version 2.6. Linux introduced a *kobject* subsystem which unifies the interfaces and therefore modules can share information in a sophisticated way.

2.4 Development Process

To get confidence that Linux can be used for safety-critical systems, the development process of Linux, the kernel, and the whole operating system GNU/Linux will be explained. To indicate the differences between the traditional way of developing software and the way the Open Source community develops software, the first section describes this mentioned traditional software development models. The discussion is based on the results of relevant literature [Gre01].

2.4.1 Software Development - The Traditional Way

A *development model* is a strategy that describes how software should be developed according to this model. It describes what different steps are necessary and when they should be done. Additionally, the model describes the phases of development and which stage follows its predecessor. Different strategies force different approaches, but the actions a developer has to do are the same, therefore it is possible to use different models for different software projects within a company without disturbing the developers.

2.4.1.1 Build and Fix Cycle

This is a model that most of the programmers have used from time to time because it is the simplest one of all. A customer (or the programmer) has an idea and starts to write a program that solves the intended task. If there are new and better ideas the programmer tries to integrate them in the program. This process stops when the program satisfies the programmer's conception of "quality management". There are no determined and separated stages of development like requirement analysis, design, or implementation. All of these steps are done by the programmer while he or she is writing the program. The advantage of this model is its simplicity and that

it can work well for relatively small software projects. There are several disadvantages of this model:

- There is a lack of documentation.
- It does not work well for large software projects.
- There is a lack of real quality management and testing.
- The programmer is the only one that knows how the code works. Therefore the project depends on the programmer.
- The code or project is difficult to maintain.

2.4.1.2 Software Life-Cycle Model

This model provides a much more structured policy which forces different steps during the development and lifetime of a system. The following steps have to be done at least once, during the lifetime of a system [PB93]:

- Requirements
- Analysis
- Design
- Implementation
- Testing
- Integration/Maintenance

Changes of the requirements can only be done during the next project and then all of the former mentioned steps are rerun from the beginning. This makes the model very strict and inflexible, because it does not provide any possibilities of going back several steps within the model. A typical software life-cycle is shown in figure 2.4.

2.4.1.3 The Waterfall Model

The *Waterfall Model* [Roy70] describes a development model where it is possible to go back one step (to the direct predecessor). This model forces several development steps, too:

- Requirements analysis
- Design
- Implementation
- Testing (validation)
- Integration/Maintenance

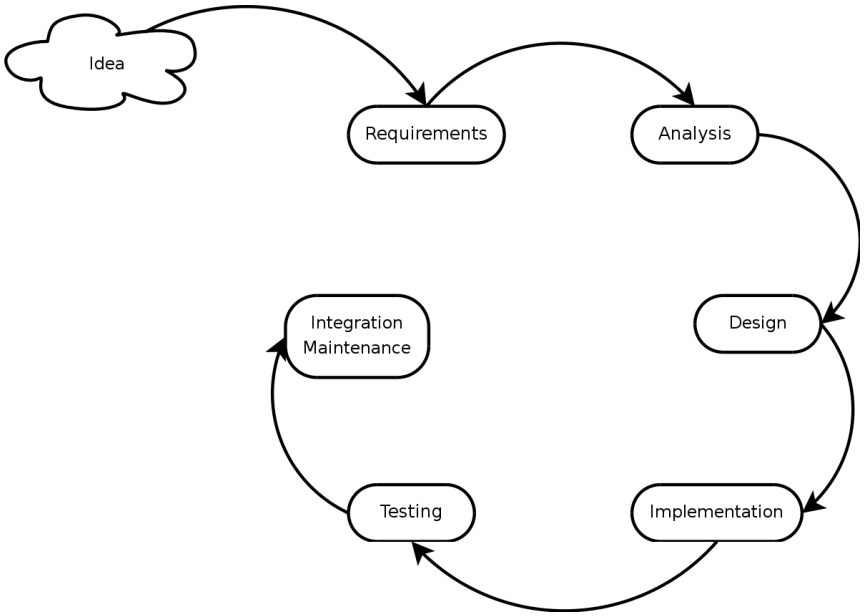

Figure 2.4: Software Life-Cycle Model

The model is an iterative approach with the constraint, that a development step is only finished, if all requirements for that step are fulfilled. This requirement should minimize the risks for the next step in the hierarchy.

The process is seen as a waterfall where the water flows downwards from one step in development to the next one (see figure 2.5).

The Waterfall Model is widely used in practice and has advantages as long as the different development steps are clearly separated from each other, therefore all possible risks have to be found at the beginning of the project. The Waterfall Model fits well if the developer team is moderately small, because then all the developers can work together on the current step. This would be very difficult for larger teams, because the number of people would be too large, and in larger teams the individual developers have different qualifications and skills.

2.4.1.4 The V-Model

This model [FM95] describes the following six steps for software development:

- Requirements analysis
- System design
- Design and implementation of modules
- Module test
- System integration
- System approval

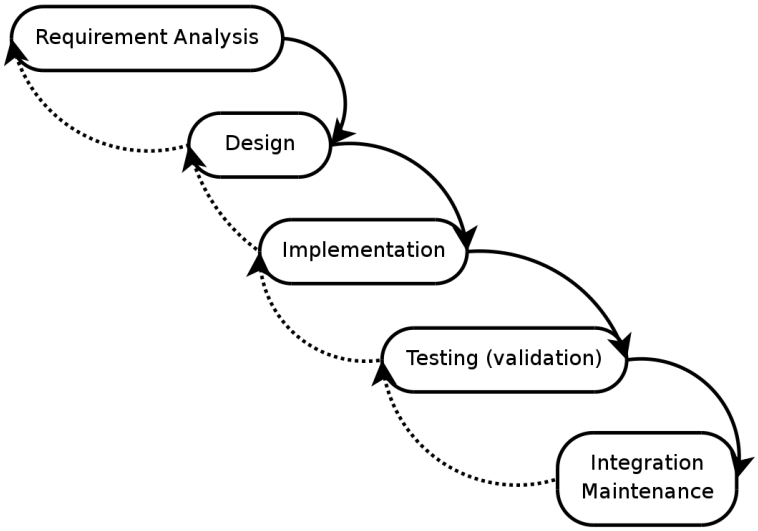

Figure 2.5: Waterfall Model

The later three steps test if the software was designed and implemented according to its specification. The finest grained level is the *module test* which tests the modules that were produced during the design and implementation stage. The *system integration* tests the correctness of the system design. At last the *system approval* tests if all requirements are satisfied. Every step in development has its corresponding test on the same level, which leads to the name for this model which is shown in figure 2.6.

Like the Waterfall Model, the V-Model forces a sequential chain of development steps. Like the two former mentioned models, it is susceptible to failures that were made in the first steps, and it makes it difficult to correct them. An advantage of the V-model is that it attaches importance to the testing process itself, it forces that every step in the development chain is tested and verified.

2.4.1.5 The Spiral Model

The *Spiral Model* which was defined by Barry Boehm [Boe88] is a system development model which pays attention to the needs of large and complex projects. The model consists of four phases which will be passed through in a cyclic manner. The spiral model is an evolutionary model where each cycle depends on the products created in the previous cycle. The following list describes the four phases of this model:

- Determine objectives, alternatives, constraints
- Evaluate alternatives, identify and resolve risks
- Develop, verify next-level product
- Plan next phases

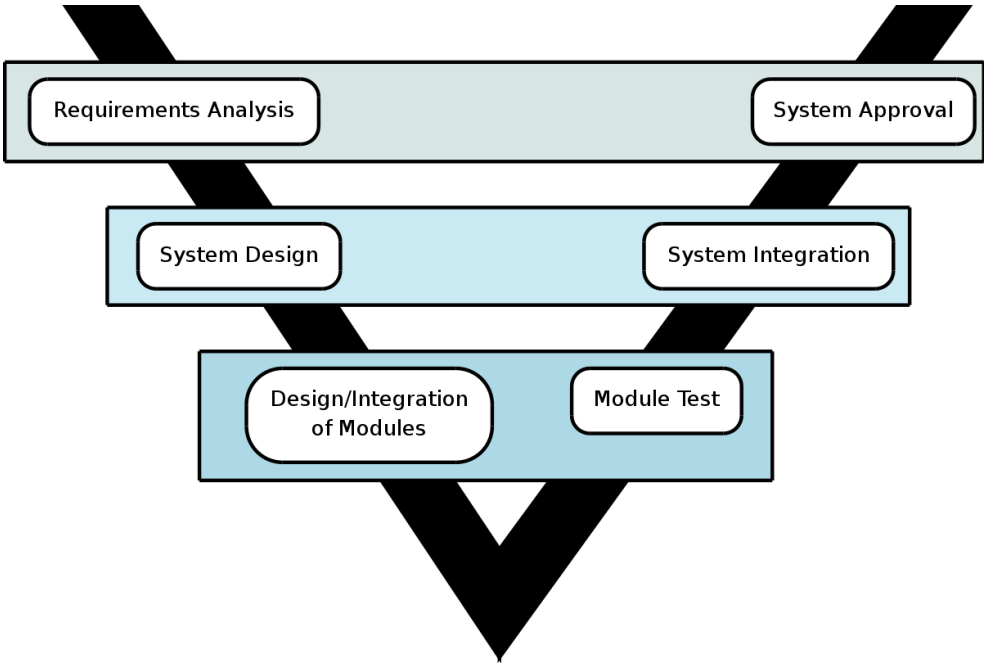

Figure 2.6: V-Model

In the first phase the goals and the resulting products of the cycle are defined. Alternative approaches and restrictions are considered, too. The second phase examines the risks that were found in the first phase and tries to find solutions for these risks. For example in this phase prototypes are built, simulation is done, or feedback from users is gathered. In the third phase the defined products are built. The number of products and the chronology of the products may vary according to the project. For example in this phase the actual coding and unit tests are done. In the last phase, the next phase is planed according to the results of the review. Figure 2.7 shows a project that was done according to the spiral model.

2.4.1.6 The Incremental Model

The V-Model and the Waterfall Model have the disadvantage that design and implementation can only start if the requirement analysis is finished. After the requirement analysis is done, it is set in stone and the implementation is done by the developers. For large projects it can take up to a year or even longer before the customer gets the finished product. It is possible that the requirements have changed during the length of time and that the product that the customer gets is not the product that he or she would like to have now.

The *Incremental Model* [Bal08] tries to solve this issues, because the implementation starts even if the whole requirements are still unknown. The key requirements have to be known before the implementation can start. The incremental model can be described in the following way:

- Software development is a stepwise process. The development is driven by the experience of the developer and the customer.
- Maintenance is seen as a new release of the product.

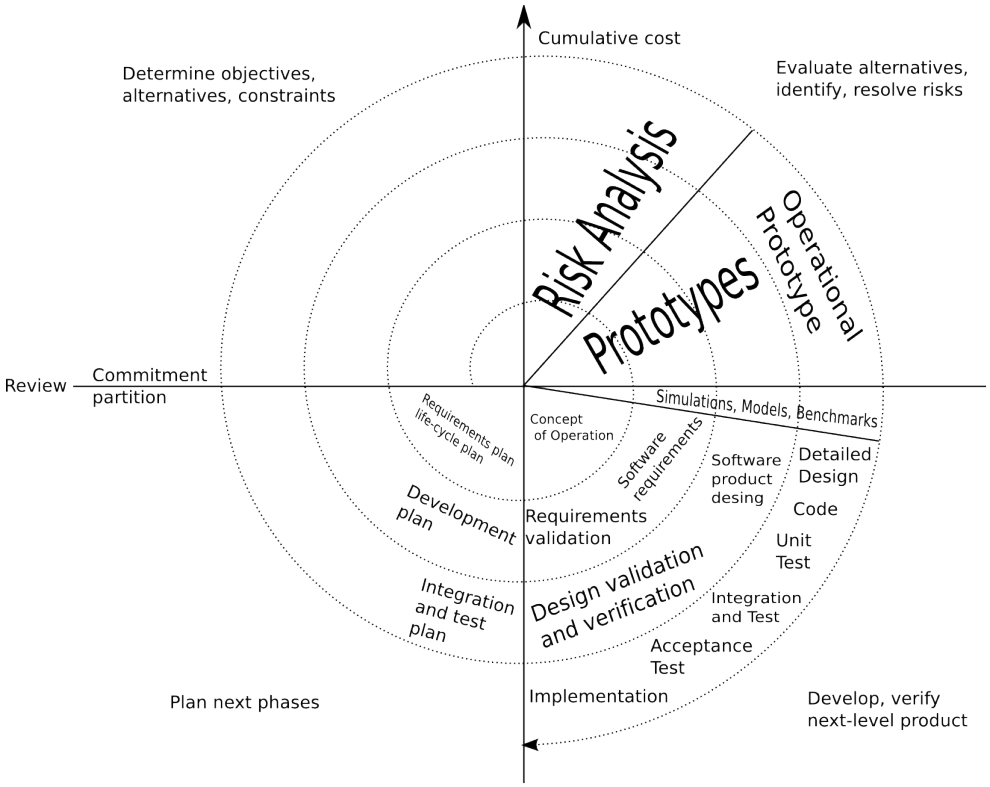

Figure 2.7: Spiral Model

- This model has the advantage that the customer does not need to know all requirements from the beginning.
- The development is code driven. The centre of interest is an executable program (or parts of it).

The Incremental Model makes it possible that parts of the product can be delivered to the customer earlier than it is possible with other development models. Even if these are only subsystems of the product. Another advantage is that changes to the requirements can be accomplished in a better and easier way.

A sample development process within the Incremental Model can be seen in figure 2.8.

2.4.1.7 Extreme Programming

Extreme Programming [Bec05] is a software development model that is relatively new and which shares ideas between the more traditional models and Open Source development. Therefore it is interesting to examine the ideas of Extreme Programming because it shows the evolution of programming concepts from old fashioned models to newer ones that are required in the digital age of the twenty first century.

Extreme Programming was invented by Kent Beck in the year 1996 while he was working as a project leader for Chrysler. The model describes several *practices* that result in Extreme

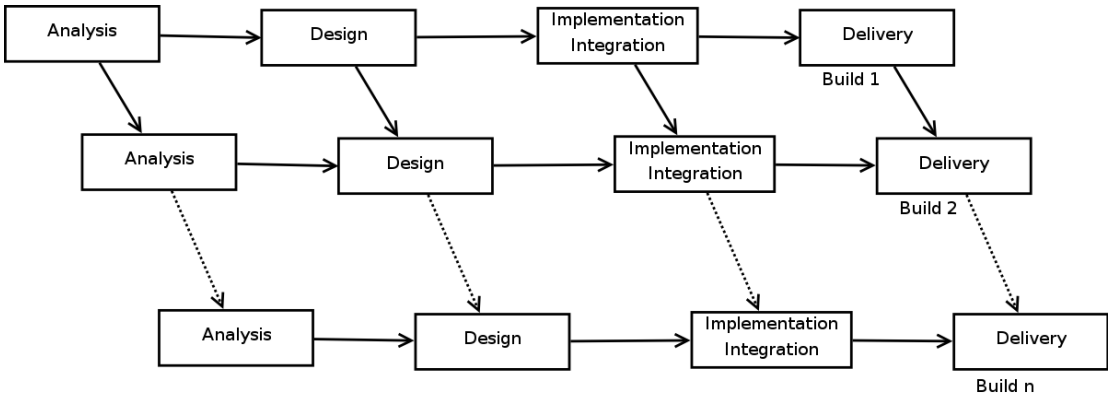

Figure 2.8: Incremental Model

Programming *values*. One advantage is the responsiveness to customer requirements. As every software developer knows, requirements change during the period of software development. Some of the traditional models do not account this problem in a proper way. For example, in models that force fixed steps, the requirements are acquired in the first step and then they are set in stone for the whole project. In contrast to this, Extreme Programming can handle such circumstances. It is important to see that at the time when this model was invented the more traditional models did not work any more in all fields of software development. The dot-com boom forced new models that could deal with fast changes of customer requirements and a short time to market.

Kent Beck [Bec05] describes the following *goals* of Extreme Programming:

- An attempt to reconcile humanity and productivity
- A mechanism for social changes
- A path to improvement
- A style of development
- A software development discipline

To achieve these goals some *values* are defined. A list of these values and their means is shown in the Appendix (table A.1).

Extreme Programming defines the following self explaining activities:

- Coding
- Testing
- Listening
- Designing

To achieve the goals, Extreme Programming defines so called *practices* which are described in table A.2 in the Appendix.

2.4.2 Kernel Development - The Open Source Way

There are some parts of the development process that can be found in the traditional way of developing software, too. For example software is reviewed and committed by developers, but the way this is done is different. Linux's development is done according to the so called *Open Source development model*. In this model the development and review is done by the public. Most of the communication between the developers is done via email on mailing-lists. There are two important tools which are used in this development process. The first one is `diff` which outputs changes to source code in a human readable way. A short example which demonstrates the use of `diff` is given. Imagine the following C-function:

```
int sum(int a, int b)
{
   return(a - b);
}
```

As it is easy to see, the function does not calculate the sum, it calculates the difference. Therefore someone will find this mistake and correct it. This can be done by everyone because everyone has read-access to the source code. This person would correct the function, and after that he or she would generate a *diff-file* which would look like this:

```
--- sum.orig.c  2008-02-06 16:30:16.000000000 +0100
+++ sum.c       2008-02-06 16:30:36.000000000 +0100
@@ -1,4 +1,4 @@
 int sum(int a, int b)
 {
-  return(a - b);
+  return(a + b);
 }
```

In our scenario, the developer would send an email with this diff-file to the kernel mailing-list. Other developers would review the patch, discuss it, and try to improve it. If the patch is good enough, it would be committed to the official source tree. For this patching the command line tool `patch` is used.

As anyone can see, this is a big advantage of the Open Source development model because a lot of people have their eyes on the source code.

On the other hand many people are afraid of the idea that everyone submits code and therefore ruins it. This expectation does not hold because it is very difficult to get own code into the official released source code. There are different layers of authorities which have to accept a piece of software before it is included in the official tree. This structure is organized like a pyramid. On the bottom of the pyramid there are hundreds of people who send their patches to a specific maintainer. Every file has its maintainer or group of maintainers. The appropriate group can be found in a file named *MAINTAINERS*, which is included in the kernel source tree. If this developer thinks the patch is correct, it gets forwarded to the maintainer of a so called sub-architecture. There are lots of these sub-architectures like "USB", "networking", "firewire", and so an. If a patch fulfils the expectations it gets forwarded to the top of the pyramid. For kernel version 2.6 this is Linus Torvalds or Andrew Morton. It is important to know that every maintainer adds a "Signed-of-by:" line to the commit log. Therefore it is easy to find the persons

who are “guilty” for non-working source code. Figure 2.9 shows the Linux development pyramid.

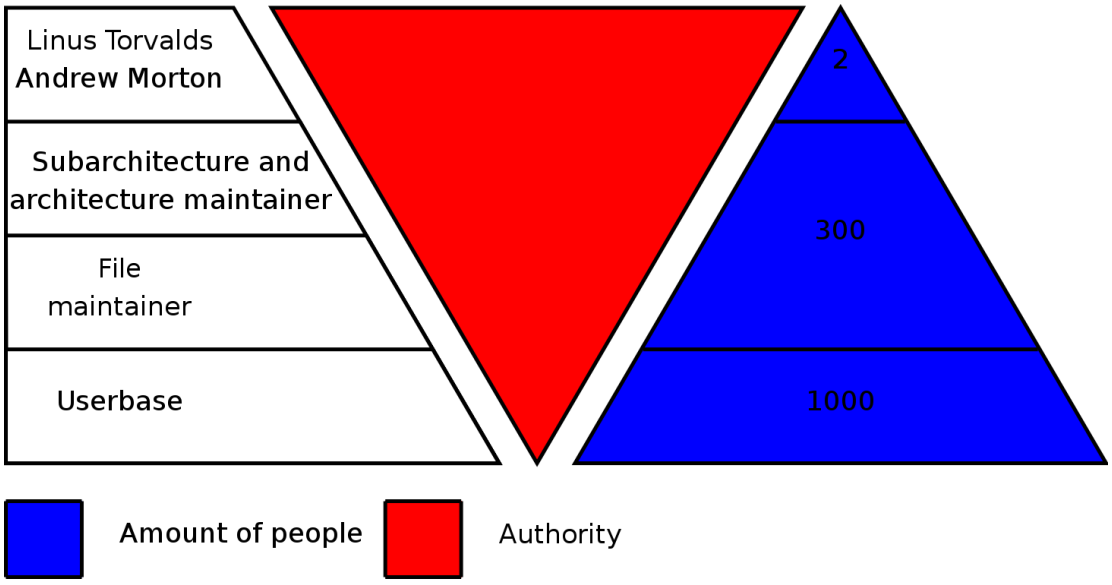

Figure 2.9: Linux development pyramid

2.4.3 Kernel Management - The git Way

A crucial part of developing software in a distributed way is the form of source code management. These methods affect the code quality and the overall quality of the project. Therefore it will be analysed how the kernel source is managed by the tool *git* and why it is appropriate for this job. This will bring confidence about *git* itself and the Open Source development model.

In the earlier days the kernel source tree was managed by a tool called *BitKeeper*, which had a client that was available free of charge. In April 2005 Bitmover, the company behind BitKeeper, decided to focus on the commercial software market and therefore dismissed the gratis client for BitKeeper. This was a historical event for the Linux community, because from one day to the other the community had the problem to manage their source without the tool which was used for years. Some developers got nervous because it took BitMover millions of dollars and numerious person years of work to develop a tool like BitKeeper.

Linus Torvalds responded the Open Source way and only two days later he presented the core of a new source code management tool called *git*. Fifteen days later it was possible to manage the current Linux source tree with *git*. Nowadays the kernel tree is still managed with *git* and it is used on a broad base for different Open Source projects.

The BitKeeper issue shows the superiority of the Open Source development model in several ways. First of all, if BitKeeper would have been an Open Source tool, the whole problem would not have existed. Only the proprietary method of software development makes it possible to take away the freedom from its users by not-releasing further versions of a software. If the source code would have been available, the Open Source community would have had the chance to continue

their work without interfering with the commercial ambitions of BitMover. On the other hand this issue shows that Open Source development can produce high quality software within very short time. Git is a prime example how free software development works. Linus released a first working component very early and the rest of the development was done by the community.

In general git is a source configuration management (SCM) tool which fulfils the same tasks as the well known *CVS* (Current Version Control), *SVN* (Subversion) or *BitKeeper*. Git is not a clone of these version tracking systems, it introduces new features which are crucial for the management of the Linux source tree. Figure 2.10 shows the typical relation between git repositories.

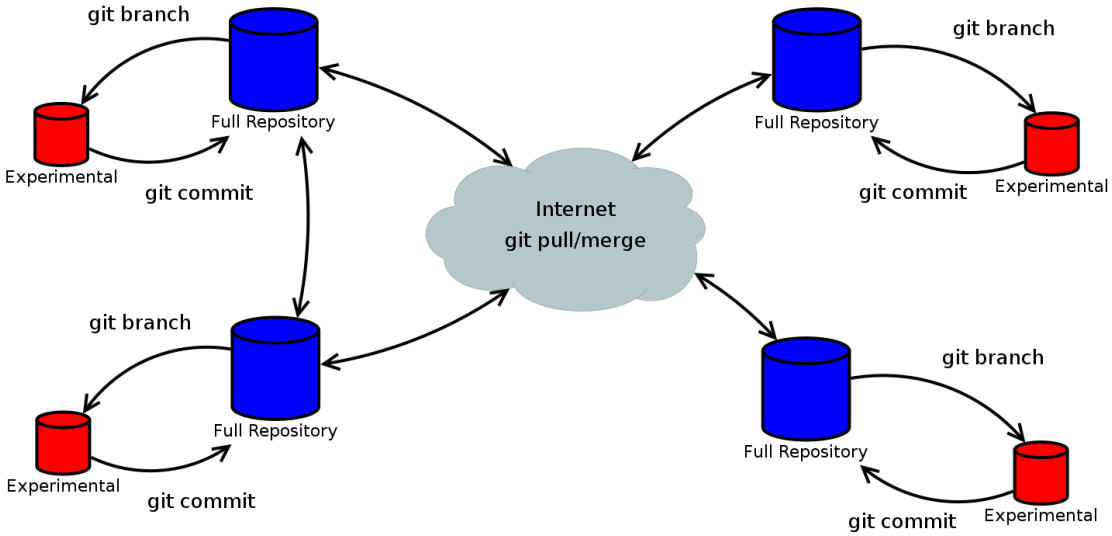

Figure 2.10: Structure of git repositories

As we have seen in section 2.4.2, the kernel development is organized in a highly hierarchic structure. There is one chief architect and his repository is the base for the others. After the individual contributors made their changes they get merged back to the master tree. Therefore a typical git cycle in kernel development would be like this (without the different layers of developer hierarchy):

- Programmer clones Linus' repository.
- Programmer makes a local branch.
- Programmer edits files.
- Programmer commits files to his or her repository.
- Linus pulls the changes back to the official tree.

One of the most remarkable features of *git* is that it guarantees source code integrity. Obviously this integrity is important because it ensures that nobody has the chance to alter source code files without recognition. If that would be possible it would be easy to inject malicious software like *backdoors*, *trojan horses* or *root kits*. By design these changes in the source code would be recognized because *git* names files by its content. This name is the SHA1 (Secure Hash Algorithm)

checksum of the file itself. Therefore *git* guarantees a high level of integrity because a user can validate the content of the file by generating the SHA1 checksum and comparing it with the name of the file. The next snippet of code will demonstrate this behaviour of naming files:

```
$ mkdir project
$ cd project
$ git init
Initialized empty Git repository in .git/
$ echo "content of my file" > file.txt
$ git add .
$ git commit -a -m "initial release"
Created initial commit 32921ff2802ab393cfe948b23f2cde99f93ba8a8
 create mode 100644 file.txt
$ echo "extended version" >> file.txt
$ git commit -a -m "added magic"
Created commit a8f23b219c7bec1238743becf12a289e1234d843
```

The commit name is defined by the following parameters:

- The content of a object.
- The "parent" commits of an object.
- The comment message for the object.

This makes it clear that the whole history of a file is represented by its checksums and that it is not possible to alter any file in the tree without recognition. The revision, a so called commit, depends on the whole development history of the project. To identify the whole development tree someone has to publish the name/checksum of the top level commit. This can be secured and checked by the use of digitally signing this message via tools like PGP (Pretty Good Privacy) or GPG (GNU Privacy Guard) [18].

One important thing to notice about *git* is that it is a total new approach of source code management. All the other widely used tools have a central repository where privileged users can commit their changes to this central repository. This has some major disadvantages because someone has to decide who is trusted to make changes to this central repository. Git uses a much more *decentralized* approach. Every developer can *clone* other repositories and work on his or her own repositories to make code changes. Technically all these repositories have the same privileges. A developer can pull back code from other developers to his own repository to get the newest patches. This is exactly the way Linux kernel development works. Maintainers pull back code from other developers at different kernel hierarchies. Code that comes up from sub-architecture to architecture maintainers and then up to Andrew Morton and Linus Torvalds ran through several levels of trusted developers. For example sub-architecture maintainer A knows that developer B writes good code and therefore he/she pulls code from B to his independent git repository. A checks the code from B and maybe applies some patches. After that an architecture maintainer C pulls back the changes from A, because he or she knows that A is a reliable developer. This goes up to the top of the Linux kernel development hierarchy until the chief architect Linus Torvalds releases a new version of Linux (the kernel).

The conclusion of this section is that Linux has a modern and secure tool to manage its source tree. Git fulfils all the needs that are claimed by the Linux kernel development process.

2.4.4 Release Cycle

One question to ask is how often should new software be released? There is one famous quote in the Open Source world which says "Release early, release often". In the early days of Linux, around 1991, Linus Torvalds sometimes released more than one new kernel version a day. Nowadays this release cycle is much slower, because the kernel size increased during the years of development, but it is much faster than commercial software releases. For example Microsoft Windows XP was released 2001 and Windows Vista was released six yeas later, in 2007. To get an overview of the typical release times, please see table 2.1.

Table 2.1: Kernel 2.6 release dates

Release	2.6.0	2.6.1	2.6.2	2.6.3	2.6.4
Date	18/12/2003	09/01/2004	04/02/2004	18/02/2004	11/03/2004
Release	2.6.5	2.6.6	2.6.7	2.6.8	2.6.9
Date	04/04/2004	10/05/2004	16/06/2004	14/08/2004	18/10/2004
Release	2.6.10	2.6.11	2.6.12	2.6.13	2.6.14
Date	24/12/2004	0203/2005	17/06/2005	29/08/2005	28/10/2005
Release	2.6.15	2.6.16	2.6.17	2.6.18	2.6.19
Date	03/01/2006	20/03/2006	18/06/2006	20/09/2006	29/11/2006
Release	2.6.20	2.6.21	2.6.22	2.6.23	2.6.24
Date	04/02/2007	26/04/2007	08/07/2007	09/10/2007	24/01/2008
Release	2.6.25	2.6.26			
Date	17/04/2008	13/07/2008			

The next important question is how are these releases managed? It is crucial to take a look on a typical release cycle to get confidence to this method of software development. During the life-cycle of the kernel there exist different kinds of so called *kernel branches*. The most important are:

- main: Version 2.6.x
- stable: Version 2.6.x.y.
- git snapshots: Daily snapshots from the git repository.
- -mm: Experimental patches by Andrew Morton.

Historically one of the most important digits was the second one, which is a *6* at the moment. *Even* digits stood for a stable kernel tree, and *odd* digits stood for developer trees. For example in times where version 2.4.x was up to date, there existed a 2.5.x tree that was the playground for the newest ideas for the next major kernel release. After the 2.5.x series was stable enough version 2.6.0 was released. During the changes in the development model there does not exist this special development tree any more. There is no 2.7.x tree, but there are lots of testing branches like the tree of Andrew Morton.

After each main release there exists a period of about two weeks where major differences to the kernel are allowed to happen. These suggestions are submitted to Linus Torvalds and most of these patches have existed in the so called "-mm" branch for a few weeks. After this period, a "-rc1" (release candidate 1) kernel is released, which is the signal that patches which change existing functionality are not accepted any more. In this phase patches which bring new functionality can be accepted, but only if they are self-contained and do not endanger the existing kernel functionality. Every time Linus Torvalds thinks the kernel tree is in a reasonable stable state for testing, he releases a new release candidate (-rcN). If the testing process is successful a new main kernel is released.

The stable kernels are released during two main releases. Their release name has four digits and they introduce very small new functionality. Their main purpose is to fix problems that have been found after a main release. Stable kernels are the best way for users to stay cutting edge without getting into troubles that experimental kernel branches could have.

One of these experimental branches are the "-mm" sources which are maintained by Andrew Morton. This is the place where experimental patches and new functionality are included and tested. These kind of kernels are not appropriate for production use. The "-mm" sources is a branch for developers.

Every day a git snapshot of Linus Torvalds' kernel tree gets released. This happens in an automated way and therefore stability should not be expected. These snapshots represent the current state of the kernel and are even more experimental than the "-rc" releases.

A whole release cycle is shown in figure 2.11.

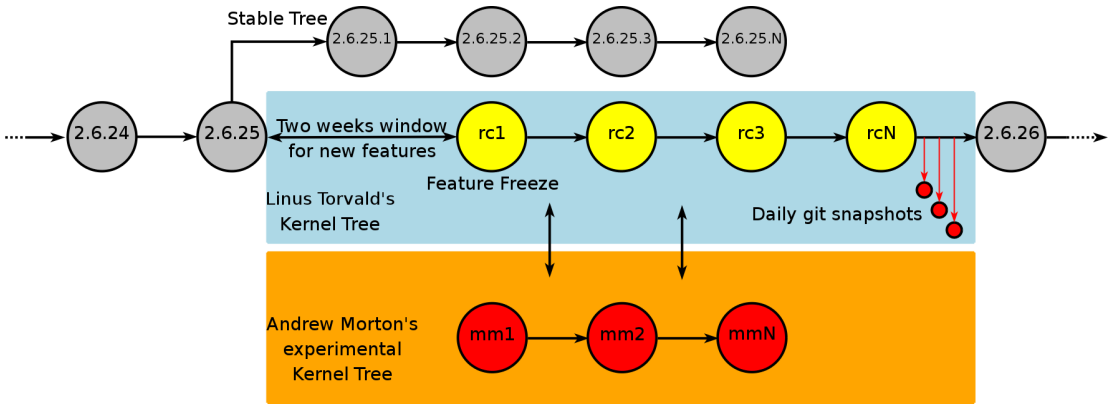

Figure 2.11: Kernel 2.6 release cycle

2.4.5 The Modularity of Linux and GNU/Linux Distributions

As mentioned before it is crucial to distinguish between Linux, the kernel, and the whole operating system. Linux is the small but important part between the hardware and the user space programs and the operating system is the union between the kernel and the user space programs. Many important programs are developed by GNU people or at least most of the free software is licensed under the terms of the GPL (GNU Public Licence). Therefore it is fair to call the whole operating system GNU/Linux and not only Linux.

The whole GNU/Linux and UNIX philosophy follows one simple paradigm, which is: "Do one job, but do it good". To fulfil this paradigm GNU/Linux and Linux share one important feature, they are all about *modularity*. This is in contrast to other operating systems like Microsoft Windows or Apple's MacOS. If someone installs a Microsoft Windows Server he or she gets all the programs that Microsoft provides to him, even if they are not needed. For example if the person wants to install a web-server, the person is forced to install a graphical user interface. Beside from the fact that the person does not need this graphical user interface (for a web-server), it is a safety and security problem. If there are exploits for the unused services, the whole mission critical system is in danger. Some of the services can be disabled but others cannot be turned off and are important for those kinds of operating systems.

At the finest grained level, the kernel level, Linux has a high degree of modularity. If someone compiles a new kernel he or she has the choice to configure it according to his or her needs. For example Linux supports about 30 different types of file systems, but if only one is necessary for the job it is easy to disable the rest of them. They are not disabled at runtime, they do not even exist in the binary image because they were disabled when the kernel was configured. A second example could be an embedded device which does not need networking features or does not have any USB or SCSI devices. It is easy to disable the whole subsystem, or to use kernel modules which can be loaded at runtime. The typical steps for configuring and running a customized kernel are the following:

- Download kernel sources and unpack them
- Configure the contents of the new kernel via a text file, a console user interface or a full featured graphical user interface
- Build the new kernel image and modules
- Configure the boot manager
- Boot the new kernel

The second level of modularity is the modularity of the applications itself. In the free software world someone has the ability to get the source code and compile the applications with exactly the features that are necessary. One example is *Apache*, the famous Open Source web-server. If someone does not need SSL (Secure Socket Layer), that someone can disable this feature at compile time.

The third level of modularity is the choice of applications. As I mentioned before, some other operating systems do not have this possibility. For example the Microsoft Windows operating system provides exactly one command line shell, the "cmd.exe", and exactly one graphical user interface, the "explorer.exe". If these applications do not accord to one's expectations, it is very difficult or even impossible to replace them. Last but not least, the user does not even has the chance to adjust these applications because they are proprietary software and the source code is not available. To revive the preceding example, GNU/Linux has a whole number of command line shells like bash, zsh, korn-shell, c-shell, tcshell, and many more.

Compiling the whole operating system is not very easy because most applications have dependencies and compiling huge packages of software consumes a lot of time. In general a fast release cycle is an advantage but it is difficult for the user of an operating system to track thousands

of packages. This is the point where so called *distributions* come into play. The first advantage of a distribution is that it supports the user while installing the operating system. It helps the user partitioning and formating the computer's hard drive, choosing the appropriate set of packages and updating the boot manager. Distributions provide package management tools, too, which make it comfortable to install new software. Different distributions choose different tools to mange their software repository. Well known is Debian's *apt/dpkg*, Redhat's *rpm* or Gentoo's *emerge/portage*. If these tools are installed, someone can type one of the following commands to install a new packages:

```
$ apt-get install apache
or
$ emerge apache
```

The second advantage that distributions provide is that someone does not need to compile the software because most distributions provide binary packages. With the use of a GNU/Linux distribution it is possible to manage all installed packages with simple commands. For example, if a user wants to check all system packages for updates, he or she can type `emerge -u system`, or `apt-get update && apt-get upgrade` and if he or she wants to check for security related updates he or she can use tools like `glsa-check` or use specific security repositories during `apt-get update`. Most distributions provide an excellent level of software quality management because for security and stability reasons it is not always the first choice to install the newest packages. For example Gentoo provides these different levels of stability during the release cycle of single packages (see figure 2.12).

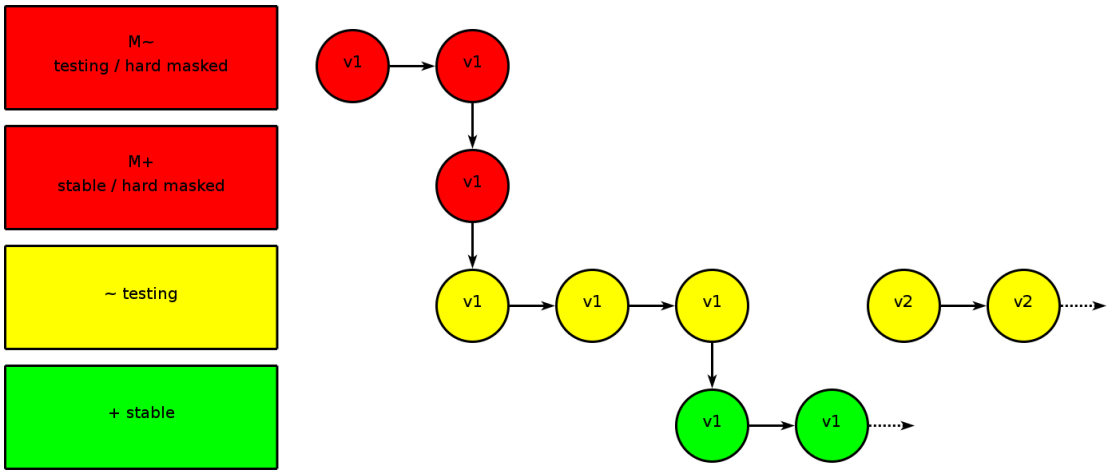

Figure 2.12: Gentoo release cycle

One big advantage of distributions is the fact that most of them provide a release cycle. The duration between two releases depends on the flavour of distribution, but release cycles make it easier for the users to get a coherent system state. A release represents a state in the time line where the distributor thinks that the packages have reached a stable state and the progress of features is huge enough that it is worth to provide a new release.

One GNU/Linux distribution that is known for its stability and well tested package base is *Debian*. Debian is a prime example for the Open Source development model. It has a *social contract*, *free*

software guidelines and a *constitution* [6]. Currently there are more than a thousand developers organized in a way that is shown in figure 2.13.

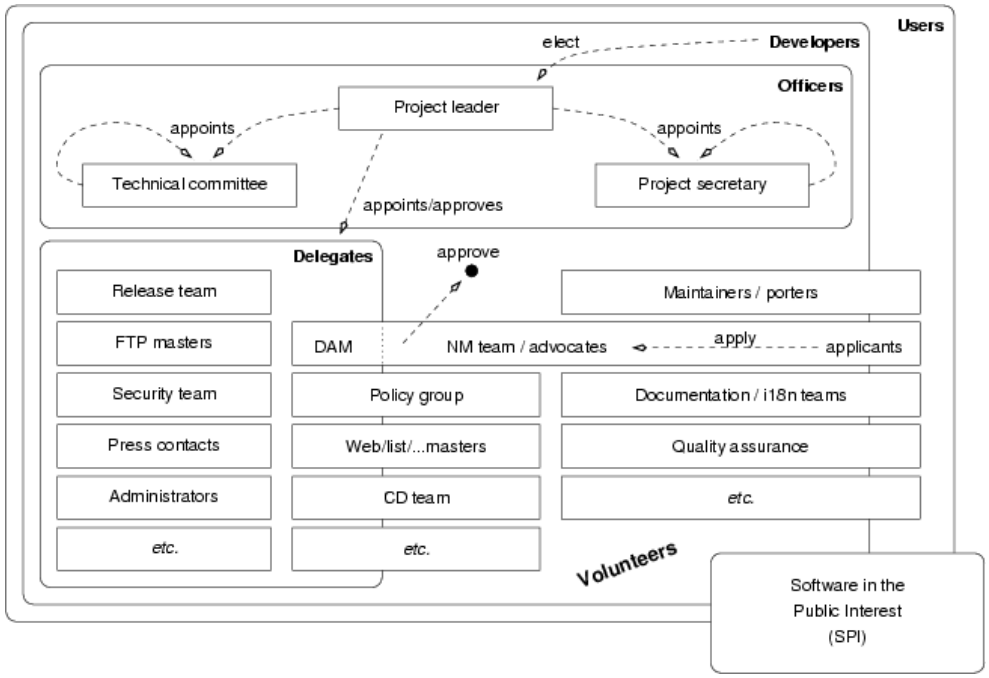

Figure 2.13: Organisation of the Debian project [Kra05]

The distribution release cycle of Debian has four main steps:

- New software is tested in an *unstable* branch.
- Software gets tested in the *testing* branch.
- The testing branch gets *frozen*.
- Generate *new release* from the frozen testing branch.

There are lots of people involved during the release of a new version of Debian GNU/Linux like the security team. The whole release process is shown in figure 2.14

The discussed layers of modularity support the argument that Linux and the whole GNU operating system provide a high level of security. A part of the overall security is gained from the modularity of Linux, the GNU software and the GNU/Linux distributions. The modularity gives the system designer the change to design it in a minimal way. Every piece of software has possible bugs, but if someone has the chance to disable whole subsystems or components, the probability to get a secure and safe system increases. Last but not least software packages in a GNU/Linux system are tested very well. The developer itself tests his or her code and after releasing the software it is extensively tested by the distribution developers. This software is much more tested than software that is just released on the internet.

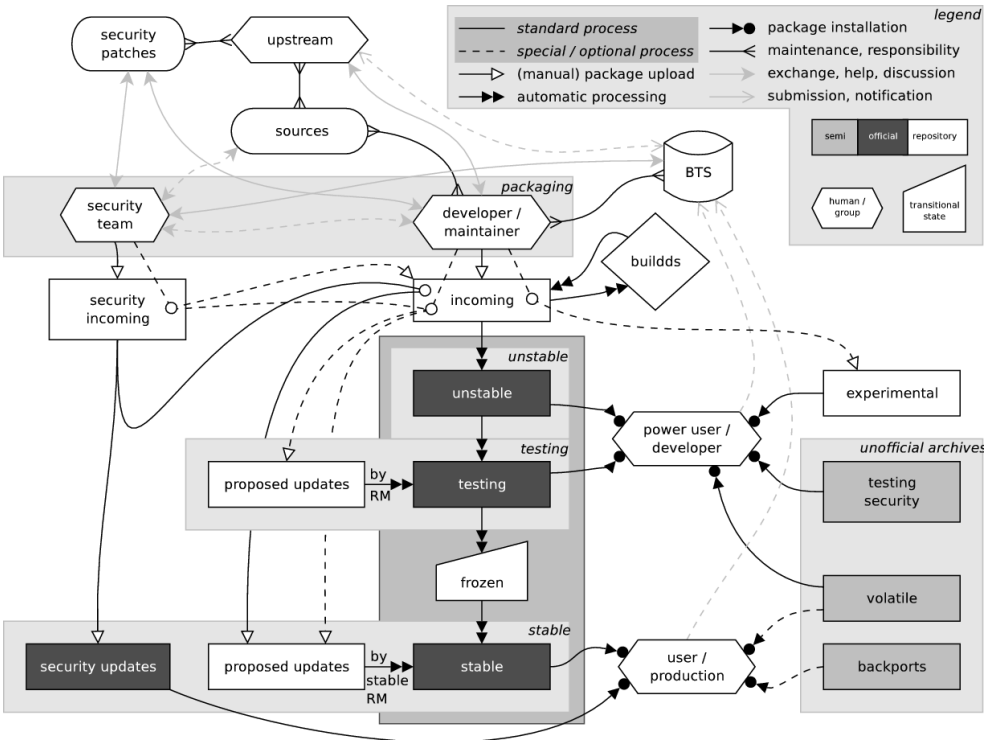

Figure 2.14: Debian release cycle [Kra05]

2.4.6 FreeBSD

GNU/Linux distributions and traditional Unices are developed and released in a different way, therefore it is interesting to analyse which approaches are taken by different open communities, that provide source code, too. For this, *FreeBSD* [10] was chosen, which is one of the best known Unices. It is an interesting excursion that shows how other operating systems that allow access to their code are released and assembled.

2.4.6.1 Goals - FreeBSD versus GNU/Linux

FreeBSD as most of the Unices of the UNIX family has different goals than GNU/Linux distributions. Both provide more or less stable release cycles, but GNU/Linux distributions concentrate on a loose coupled set of individual software packages. If someone installs a GNU/Linux distribution he or she mostly gets all the software that was available at that instant in time. For example there are different kinds of command line shells or different kinds of ftp-server software on a typical GNU/Linux distribution. On the other hand FreeBSD tries to provide a whole coherent operating system that fulfils several tasks. FreeBSD's main purpose is a stable and secure server platform. In the base system FreeBSD provides one tool for one task. For example there is one MTA (mail transfer agent), which is *sendmail*. It is possible to install other MTAs like *postfix*, but only sendmail is in the basic release set. If there is a security hole in *sendmail*, the FreeBSD developers provide the necessary patches to remove this issue. On the other hand the operating system keeps highly extendible via the *ports* system, which is a collection of *Makefiles* and additional patches that provide build information for compiling a piece of software. The BSD ports

are similar to the concept of Gentoo's *portage*, and in fact the idea of portage was taken from the BSD ports system. FreeBSD provides modern tools to mange the fetching and updating of the ports tree and for installing and updating software via the ports tree. If compiling software from source is not an option or unwanted, binary packages can be managed by *pkg-tools*. For example to install the latest *GNU Compiler Collection* (GCC), someone can use the following commands.

```
$ # install a binary package, fetched from the internet
$ pkg_add -r gcc44
$ # install from ports, the traditional way
$ cd /usr/ports/lang/gcc44/ && make install clean
$ # install from ports, the modern way
$ cd /usr/ports/lang/gcc44/
$ portinstall $PWD
```

2.4.6.2 Development Process

FreeBSD is developed in a very open way, too. The repository is readable by everyone via anonymous, read-only CVS and nowadays additionally SVN. Write accounts are not permitted to everyone, because that would cause total chaos. Write access to the CVS tree is permitted to a selected group of about 300 people which are called *committers*. The next upper level of hierarchy is represented by the *core team*, which is responsible for major decisions in which direction FreeBSD should be developed. At the moment this *core team* consists of nine people.

The FreeBSD CVS tree, which has its primary repository in Santa Clara CA, has two major development branches which are called *FreeBSD-CURRENT* and *FreeBSD-STABLE*. The *current tree* is the place where new features are introduced and tested by the developers and the user community. It is the playground for new ideas and therefore it should not be used on productive systems. On the other hand there is the much more stabilized *stable tree* where well tested software is merged into the CVS tree. New software first enters *current* and after the testing process it is merged into *stable*. At some instants of time there are releases of FreeBSD, which are generated from the *stable tree*.

2.4.6.3 Release Process

FreeBSD tries to provide a stable and fixed release scheme which ends in a new release every four months. New releases are built from the "-STABLE" tree and according to FreeBSD's developers the effort begins to rise about 45 days before the anticipated release date [11]. At this point in time, the developers are informed that they have 15 days time to merge software changes until the so called *code freeze*. The developers now start to merge code from the "-CURRENT" tree to the "-STABLE" tree. The next important step to a new release is the *code slush* which starts about 30 days before the planed release. From now on there are about 15 days where the source code has to be approved by the *Release Engineering Team*. Only a limited set of operations is allowed to the source code which includes the following changes [11]. See figure 2.15 for details.

- Bug fixes
- Documentation updates

- Security-related fixes of any kind
- Minor changes to device drivers, such as adding new Device IDs
- Any additional change that the release engineering team feels is justified, given the potential risk

During the last 15 days before the release the developer team tries to release *release candidates* on a weekly basis.

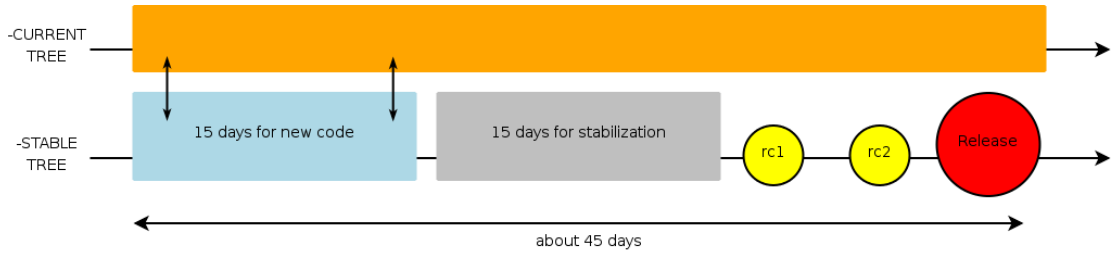

Figure 2.15: FreeBSD release cycle

2.5 Testing, Standardization and Backports

Testing and standardization are indispensable processes that have to be done for operating systems in a safety-critical context. A description how Linux is tested and verified is given and a selection of projects that exist for fulfilling this important issue is shown. Without these methods of testing and standardization a kernel cannot be used for safety-critical systems.

2.5.1 Testing and Standardization

A huge part of the testing efforts is done by the community and the Open Source development process itself. This development model itself guarantees a high degree of quality management because hundreds of people have their eyes on the Open Source code. This model, which was described in section 2.4 is relatively new and sometimes in contrast to the traditional methods of software quality management. These differences do not have a negative impact on the product itself, they provide several advantages. It is obvious that the more people review a specific code the higher the probability is that a bug is found. It would be impossible or extremely expensive if a company would hire a developer-team that has the manpower of the Linux kernel developer team. Therefore a high level of stability, safety, and security is reached by the Open Source development process itself, especially when this development is managed in a sophisticated way like it is done by the Linux community.

On the other hand the development of safety-critical systems and the corresponding standards (chapter 3) force some degree of formal verification and testing, which is necessary to guarantee safety. The industry has reacted to this important problem and several kinds of organisations and projects exist that try to provide this level of traditional software quality management. In

this thesis the *Linux Test Project* (LTP), the *Linux Foundation*, the *Linux Kernel Performace* project, and the *AutoTest* project are mentioned.

The *Linux Test Project* was founded by SGI and is maintained by IBM. The main goal is to provide a *test suite* to validate the reliability, robustness, and stability of the Linux kernel [30]. At the moment this suite contains more than 3000 tests. Most of the project is written in ANSI C (94%), the rest in Bash-scripts and Perl-scripts. The test suite provides an API, too, and therefore it is possible to extend it with user defined tests that are necessary for special safety-critical systems. The suite itself contains tests for the following purposes:

- File system stress tests
- Disk input/output tests
- Memory management stress tests
- Inter Process Communication (IPC) stress tests
- Scheduler tests
- Commands functional verification tests
- System call functional verification tests

The test suite is comfortable to use, for example there exists a `alltests.sh` shell script which executes all the tests in the suite. To examine the code coverage, which represents the code that is executed, the tool *gcov-kernel* exists. It is a extension to the GNU gcov tool and provides coverage support for the kernel and a interface to to the `/proc` file system by a kernel module named *gcov-proc* As frontend the traditional GNU gcov tool, which is part of the GCC (GNU Compiler Collection), can be used. Gcov provides information about how often a specific line of code has been executed by the test suite. This tool is already used for traditional software code coverage analysis and the patches and kernel modules from the Linux Testing Project make it possible to use gcov for kernel level analysis.

For visualization of test results there exists a second extension to GNU gcov namely *lcov*. This package includes several Perl-scripts that visualize the code coverage with coloured bars and HTML output format.

The second and definitely one of the most important projects for Linux standardization is the *Linux Foundation* (LF). The Linux Foundation is a merge from *Open Source Development Labs* (OSDL) and the *Free Standards Group*. Beside from employing Linux developers like Linus Torvalds and therefore guaranteeing that key developers are able to continue their work, the Linux Foundation is a platform for standardisation and collaboration [26]. The LF offers certification according to several versions of the Linux Standard Base (LSB). This standardization is important to guarantee that different GNU/Linux distributions share the same standard base and that therefore a LSB certified application will work on every LSB certified distribution. If an application or distribution passes the certification process it is permitted to use the LSB trademark. The current version of the LSB is 3.2 and is defined in two parts. The first one is the architecture independent part, the second is architecture dependent. Each of these parts contains different modules, which are “Core”, “C++”, “Desktop”, “Interpreted Languages”, and “Printing”. Table 2.2 provides information about the main categories and their modules.

Table 2.2: Linux Standard Base version 3.2

LSB 3.2	Modules
Executable And Linking Format (ELF)	Low Level System Information, Object format, Dynamic Linking
Base Libraries	Program Interpreter, Interface Definition / Data Definition / Interfaces for libc, libm, libpthread, libgcc_s, libdl, librt, libcrypt, libpam
Utility Libraries	Interface Definition / Data Definition / Interfaces for libz, libncurses, libutil
Commands and Utilities	Commands and Utilities, Command Behaviour
Execution Environment	File System Hierarchy, Localization
System Initialization	Cron Jobs, Init Script Actions, Comment Conventions for Init Scripts, Installation and Removal of Init Scripts, Run Levels, Facility Names, Script Names, Init Script Functions
Users & Groups	User and Group Database, User and Group Names, User ID Ranges, Rationale
Package Foramt and Installation	Package File Format, Package Script Restrictions, Package Tools, Package Naming, Package Dependencies, Package Architecture Considerations

At the moment, the Linux Foundation consists of an elected chair of 9 people from leading Open Source software projects, companies, and distributors like MySQL, IBM, Intel, Sun, Usenix, RedHat, Novell, The Open Group, VMWare, Nokia, and many more.

The *Linux Kernel Performance* project [22] is maintained by five dedicated kernel programmers and tries to test the kernel in a systematic and disciplined way to improve the performance of the kernel. The testing itself is done by running a large set of benchmarks covering different kernel core components [22]. The tests are run on different hardware platforms every week and the results are published on the project page. Everyone of these five project maintainers is associated with the Intel company and therefore the tests are run exclusively on Intel hardware. The kernel under test is Linus Torvald's latest git snapshot (see section 2.4.4).

According to the project page the following components are tested:

- Raw server performance and network scalability
- Micro benchmarks for measuring the key system performance
- Network performance tests (TCP and UDP, DLPI, UNIX Domain Sockets,...)
- Memory and disk operation, virtual memory management
- I/O workload tests: Sets of files are created and operations are performed on them
- Filesystem tests (read, write, re-read, re-write, read backwards, fread, fwrite, random read,...)

- CPU intensive benchmarks

The last project presented in this thesis is *test.kernel.org* [23] (TKO). This project focused on a whole test environment called *autotest* for the kernel which makes it possible to test Linux in a highly automated way. The present development has shown that manpower is more expensive than machine-power and therefore a automated test approach fulfils the needs of the kernel community best. Software companies can afford the costs of a quality management team, which is much more difficult in the Open Source world. Therefore it is a better approach to focus on the wide range of machines instead of expensive programmers. There are other arguments that support the need for automated testing. For example Linux supports a broad spectrum of hardware and many different combinations of them. The more hardware that is supported and the more portable a kernel is, the more important an automated testing approach becomes. Open Source is an evolutionary and fast process. Developers join projects but some of them leave it as fast as they came in and therefore it is a good idea to keep their knowledge in the (testing) system instead of keeping the knowledge in the person itself. Automated testing speeds up the testing process because the developers get feedback very soon when their code is still in their minds. An additional advantage is that most of the developers have a limited range of different hardware available, but Linux supports a broad range of hardware. For these developers it would be very hard to get their hands on different and not so wide spread hardware. Therefore feedback from TKO is highly important for them. An extended discussion of the advantages of an automated testing approach can be found in the pager "Fully Automated Testing of the Linux Kernel" [BW06].

2.5.2 Backports

GNU/Linux distributions are able to provide a stable system with long term support without breaking the system stability and without introducing new bugs. This introduces a longer life-cycle which is important for safety-critical systems, too.

From the kernel side this is managed via the so called *stable tree* (section 2.4.4). This tree, which has four digits (2.6.x.y) does not try to introduce new features, it is for updates concerning safety and security of the system. This is necessary, because new features always have the potential of introducing new bugs. Therefore concentrating on fixing bugs without introducing new features is the best way to keep the system current and stable. New kernel releases sometimes provide changes in the API which could break the compatibility to existing third party modules, but if someone stays in the stable tree he can avoid this nasty side-effects.

On the other hand the operating system provides much more software than the kernel and therefore it is important that COTS providers have mechanisms to guarantee a stable system, too. Some GNU/Linux distributors use so called *backports* for this purpose. Most software writers and software projects concentrate on their single project, but they do not take direct actions to integrate their project into a coherent operating system (GNU/Linux distribution). Therefore a second layer of developers exist who try to integrate the individual software packages in the coherent operating system. These programmers actually do not work on the software package itself, but they work for the GNU/Linux distributors. If a new release of a package is done, they try to provide new packages for their distribution. If there are critical fixes, some distributions provide back-ports to their stable line of distribution. For example, imagine a distributor who released version 1.2.3 of a software for its stable system. If there exists a security hole in this

version, the original authors of the software will provide version 1.2.4. But sometimes the developers introduce new features (and therefore the possibility of new bugs), too. The distributor cannot switch easily to this version. Maybe the 1.2.4 release has a changed API and therefore all packages that depend on it would have to be updated, too. Therefore the distributors provide a *backport* of the security fixes for their old and stable version 1.2.3 without the new features. This package could be named 1.2.3_1 or something similar.

In conclusion, both, Linux and GNU/Linux distributions provide mechanisms for a stable, safe and secure operating system. This is done via the *stable tree* of the kernel and the *backports* form the GNU/Linux distributors.

3 Safety

A clear definition of the term *safety* is required to decide if an operating system can be used in a safety-critical context. It will be delimited from terms like *reliability*, *availability*, and *maintainability*. After that the focus is on the question if it is possible to certify software like Linux according to *CENELEC* standards which are in use in the railway domain. The last section examines if COTS (Commercial Off The Shelf) components can be used in safety-critical applications and if there exist appropriate GNU/Linux components for this task.

3.1 Definition of Safety

The definition of *safety* starts with two quotes, one from Hermann Kopetz and one from Avizienis, Laprie and Randell. "Safety is reliability regarding critical failure modes" [Kop97] and "Absence of catastrophic consequences on the user(s) and the environment" [ALR01]. The first quote depends on the definition of reliability, which will be discussed in detail in section 3.2, but it gives a clear statement, that a system is only safe, if the user can place trust in the system that it fulfils its specification according to critical failure modes. The second quote includes a more practical relation to the environment that is controlled by a safety-critical application by saying that these systems are not allowed to do harm to users or the environment. The conclusion of these two quotes is, that a system or application is only entitled as *safety-critical* if it controls a process that has a potential risk for its environment or users.

The question if it is necessary to provide safety has two components. The first one is the ethical component, the second one is an economic question. Sometimes the economic overrules the moral as we can see in the following example of immoral cost-benefit analysis. In the early 1970's the Ford Motor Company released a new line of cars, the *Ford Pinto*. Before the car was released several crash tests proved, that the car has serious design flaws [Str80]. The problem was the design and the location of the gas tank that caused fire or an explosion during a rear end collision at an even very low impact speed. This issue was known, but Ford did not react until forced by law. The costs to save lives was about eleven dollar per car, but from a cost-benefit standpoint it was cheaper to let people die.

> "A cost-benefit analysis prepared by Ford concluded that it was not cost-effecient to add an $11 per car cost in order to correct the flaws. Benefits derived from spending this amount of money were estimated to be $49.5 million. This estimate assumed that

> each death which could be avoided would be worth $200,000, that each major burn injury that could be avoided would be worth $67,000 and that an average repair cost of $700 per car involved in a rear end accident would be avoided. It further assumed that there would be 2,100 burned vehicles, 180 serious burn injuries, and 180 burn deaths in making this calculation. When the unit cost was spread out over the number of cars and light trucks which would be affected by the design change, at a cost of $11 per vehicle, the cost was calculated to be $137 million, much greater then the $49.5 million benefit". [5] [Str80]

Safety does not only have moral implications, improper system and safety design can lead to enormous costs. One well known example is the *Ariane 5* accident [Lio96]. The rocket had a 500 million dollar set of four identical scientific satellites on board which were lost during the accident. Another example is the *Therac-25* accident [Lev95] which happened between 1985 and 1987. The Therac-25 system was a medical accelerator which had malfunctions and it delivered lethal X-ray doses at several medical facilities. The system provided two different kinds of radiation, a low-power electron beam and in additon X-rays. Due to internal race conditions, Therac-25 radiated the high-power X-rays with the metal X-ray target out of position. At least 5 people died, several others were injured.

These examples show that safety is an important issue in the modern world and it leads to the question how safety can be measured. In industry, standards are used to describe the development of safety-critical systems. Software programs are not like mathematical formulas and as it is known from several fields of computer science like *testing* or *quality management*, it is impossible to *prove* that a piece of software is correct in a mathematical sense. Therefore someone must find *arguments* for the safety of a piece of software that confirm that it is *unlikely* that the software fails. Certification and standards provide mechanisms that can be used to find these former mentioned arguments. For example standards describe how software should be tested. If it is enough to black-box test the software or if additional white-box tests are required. Therefore a system developer who *trusts* a specific standard has trust in a piece of software that was certified according to this standard.

RCTA/DO-178B [ARI92] is a de facto standard for the development of safety-critical avionics software. According to this standard, systems have to be certified as a whole, hardware and software. One of the most important requirements forced by DO-178B is the *traceability*. "The evidence of an association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation".

On the other hand there exist several European standards like IEC 61508 [IEC98, FG] where it is possible to certify *individual components* of the safety-critical system. The criticality of a system function is categorized in so called *Safety Integrity Levels* (SIL). There exist four (1-4) different levels of integrity. Each level forces different development and verification methods. Table 3.1 shows the different levels of severity in DO-178B and IEC 61508. Note that this table is not a side to side comparison of these two standards. For example a "Level B" system has completely different requirements than a "SIL3" system.

According to [Kop97] these levels of severity have the following impact on the environment:

- Catastrophic: A fault that prevents continued safe operation of the system and can be the cause of an accident.

Table 3.1: Level of failure severity

Severity of Failure	DO-178B	IEC 61508
Catastrophic	Level A	SIL 4
Hazardous/Severe	Level B	SIL 3
Major	Level C	SIL 2
Minor	Level D	SIL 1
No Effect	Level E	—

- Hazardous: A fault that reduces the safety margin of the redundant system to an extent that further operation of the system is considered critical.
- Major: A fault that reduces the safety margin to an extent that immediate maintenance must be performed.
- Minor: A fault that has only a small effect on the safety margin. From the safety point of view, it is sufficient to repair the fault at the next scheduled maintenance.
- No Effect: A fault that has no effect on the safety margin.

3.2 Dependability

It is essential to get a clear view and definition of the terms *reliability*, *availability*, *maintainability*, and *safety* (RAMS), therefore the following section gives the required distinction of these terms. There are several interconnections between these terms which will be discussed, too. A general term, with several additions like *security*, or *survivability* is referred as *dependability*. Dependability of a computing system is the ability to deliver service that can justifiably be trusted [ALR01]. Figure 3.1 shows the concept of dependability according to its *threats*, *attributes*, and its *means*.

3.2.1 Threats

The threats for dependability can be described by the so called fault-error-failure-chain which is shown in figure 3.2.

The *fault* is the starting point of the chain and it is a circumstance that causes an error. Faults can be categorized by *nature*, *perception*, *boundaries*, *origin*, and *persistence* [Kop97], which is shown in figure 3.3.

- Nature: The nature of a fault can be by *chance* or *intentional*. For example ageing of an electronic component is a fault by chance whereas improper usage of a system can happen by intention. For example an angry user who hits the monitor.
- Perception: A fault can be caused by a *physical* problem like a bird that flies through a wireless communication signal or it can be by *design*. Design faults can happen by inappropriate specification of the system or by inappropriate implementation by the programmers.

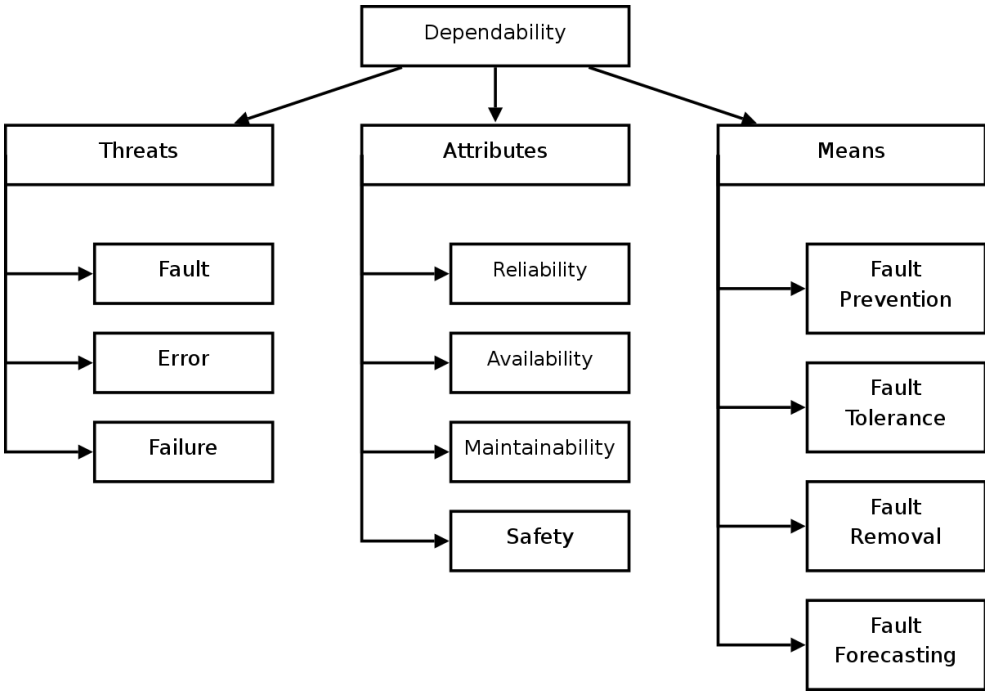

Figure 3.1: Dependability according to threats, attributes and means [ALR01]

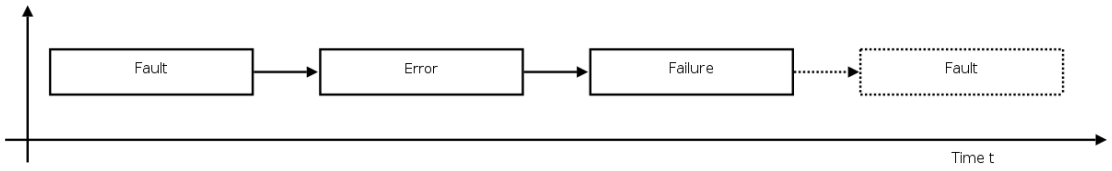

Figure 3.2: Fault, error, failure chain [Kop97]

- Boundaries: *Internal* faults happen if the fault is contained in the system itself, whereas *external* faults are caused by external influences like over voltage.
- Origin: *Development* faults are caused by the system development, like misunderstanding the task of the system or bad design decisions. *Operation* faults can appear if the user of the system is not able to handle it in a sufficient way.
- Persistence: *Transient* faults disappear without any influence on the system by them self, whereas *permanent* faults do not disappear and therefore force some form of intervention.

If faults occur they can lead to an *error*. An error is an internal, unintentional *state* of the system. Errors can be *transient* or *permanent* [Kop97].

- Permanent: This class of errors need a correction mechanism to be removed.
- Transient: Transient errors occur from time to time, caused by a fault, but they disappear without causing a failure.

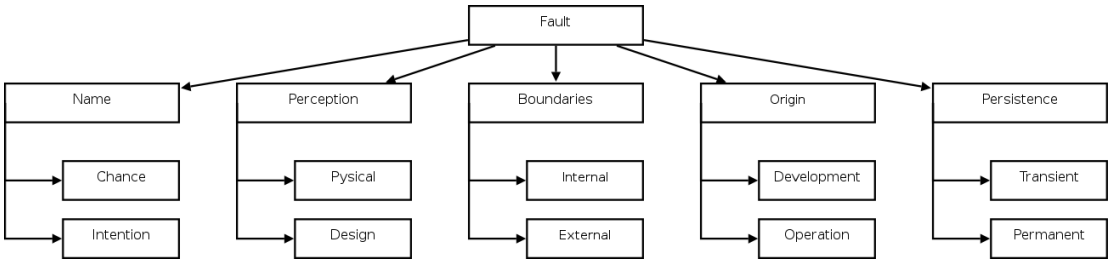

Figure 3.3: Categorization of faults [Kop97]

If the output of the system does not conform to its specified behaviour it is called a *failure*. According to Laprie [Lap91] and Kopetz [Kop97] failures can be classified by *nature*, *perception*, *effect*, and *oftenness*, which is shown in figure 3.4.

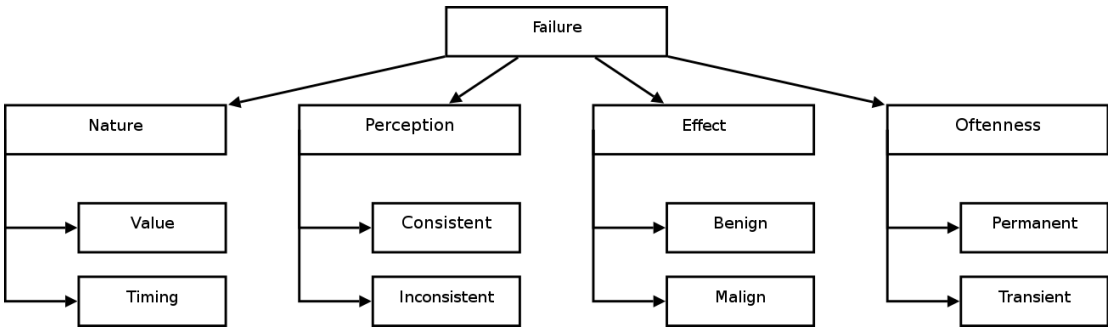

Figure 3.4: Categorization of failures [Kop97]

- Nature: A failure in the *value* domain is a failure like a bit-flip where a zero flips to a one. The second class are *timing* failures. They occur if a correct value is too late or too early in a point of time.
- Perception: If the failure is *inconsistent*, different users or systems have a different view of it whereas a *consistent* failure is seen by all users in the same way. An example is a data failure, where all the users see the same (wrong) result.
- Effect: *Benign* failures do not cost more than the component itself, whereas *malign* failures cost much more. A failure in a railway control system can cost much more than the system itself, in the worst case it can cost the life of the passengers. If a system has the potential of malign failures, these system is *safety-critical*.
- Oftenness: If the system is able to operate after a failure occurs, it is called a *transient* failure, whereas *permanent* failures need some form of correction.

3.2.2 Attributes

The attributes of dependability are *availability*, *reliability*, *safety*, and *maintainability*.

Availability describes the readiness of a system to deliver its specified behaviour. Two important values in the context of availability are the *Mean Time to Failure* (MTTF) and the *Mean Time to Repair* (MTTR). Sometimes the sum of MTTF and MTTR is referred as the *Mean Time between Failures* (MTBF). If the availability gets assigned the letter 'A', it is described by the following formula: $A = \frac{MTTF}{MTTF+MTTR} = \frac{MTTF}{MTBF}$. The interaction of these terms is shown in figure 3.5.

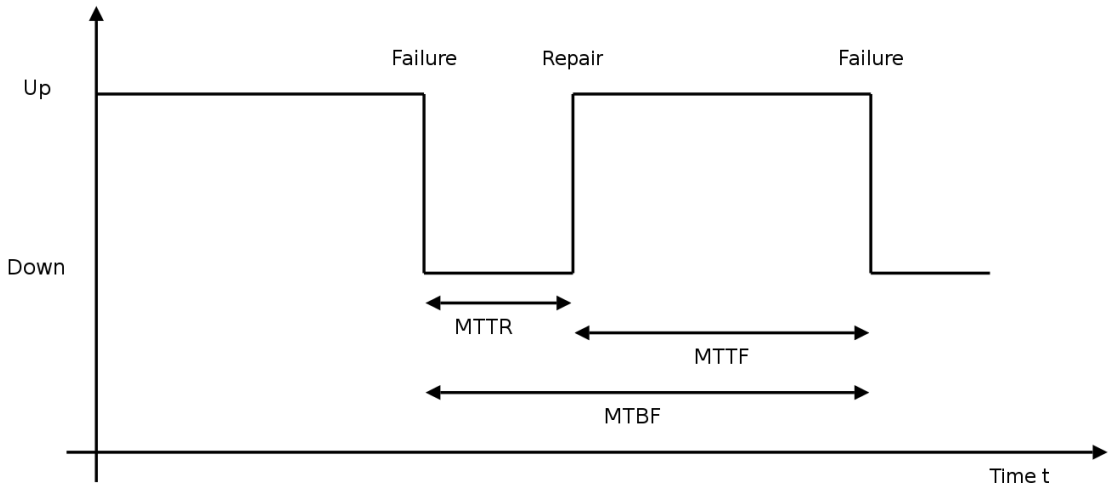

Figure 3.5: MTTR, MTTF, MTBF [Kop97]

The term *reliability* describes the continuity of correct service. It is the probability that a system, that worked correct at time $t = t_0$ will deliver this correct service until time t. λ is the constant failure rate (typically per hour). The reliability is given by the following formula: $R(t) = e^{-\lambda(t-t_0)}$. The inverse of the failure rate ($\frac{1}{\lambda}$) is the *Mean Time to Failure* [Kop97].

Maintainability of a system is the probability that a system can be repaired after a failure within a given duration in time. One measure for maintainability is the *Mean Time to Repair* (MTTR).

3.2.3 Means

The means of dependability are *Fault Prevention*, *Fault Tolerance*, *Fault Removal* and *Fault Forecasting*.

The term *fault prevention* deals with methods of avoiding the introduction of faults. This prevention is done by methods of software or hardware quality management. For example some standards force strict design and programming paradigms. The well known *V-Model* or the *Waterfall Model* can be used to guarantee a healthy software life-cycle. Modularization of software is a method of fault prevention, too. Individual components have less complexity and they can be tested easier. In the field of hardware the choice of high-quality suppliers or the enforcement of strict design rules prevents the introduction of faults. Even obvious actions like better shielding can be used to improve the overall system quality.

Fault tolerance is a method of tolerating individual faults but keeping the whole system alive. This fault tolerance is achieved by *redundancy* in the system. For example if one component fails an other redundant component takes over the job of the failed one. There are different classes of

failures, which force a different level of redundancy [PSL80]. In the following examples k is the number of *failed* components [Kop97].

- *Fail-silent* behaviour is given if the system produces correct values or it does not produce any values at all. This class requires $k + 1$ components in total, and therefore 1 extra redundant component.
- If a failure is *fail-consistent*, all users or the environment see the same result. All users have a coherent view of a value or an output, but this does not mean that this output is correct. If a component is fail-consistent, $2k + 1$ components are required to cover k faulty components.
- If the failure is *inconsistent*, also referred as *Byzantine failure* all the users have a different view of the value that is produced by the failed component. To tolerate such k failed components, $3k + 1$ components are needed.

There are several ways fault tolerance can be achieved, but all of them deal with redundancy. One way is to add redundancy to the transmitted *information*. For example a Hamming-Code could be added, which allows the reconstruction of the information after a flopped transmission. Another approach is to add redundancy in the *time* domain. For example every message could be sent a second time if it is necessary without violating the timing of the whole system. The third approach is *physical redundancy* which can be done in software or in hardware. If it is done in software, critical processes exist more than once and if one of them fails there are still enough processes that can cover the job of the failed one. For safety-critical systems redundancy is often added by additional hardware. One often used method is *Triple Modular Redundancy* (TMR), which guarantees a high degree of fault tolerance. The behaviour of a TMR-system is shown in figure 3.6.

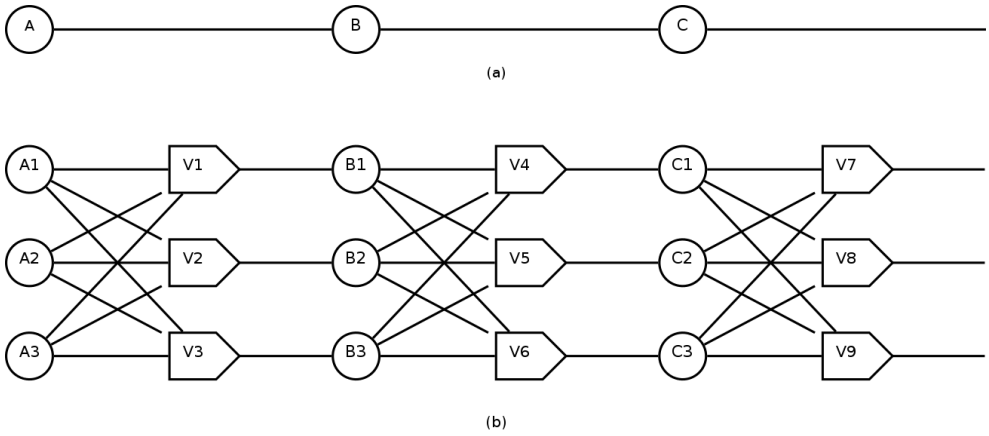

Figure 3.6: Triple modular redundancy

The system [Tan03] in figure 3.6 (a) consists of 3 individual subsystems and does not have any redundancy built in. If one of the subsystems fails, the whole system fails. On the other hand every component of the system in figure 3.6 (b) is replicated three times. In addition there are *voters* in each stage of the system. Each voter has three inputs and one output. The output of

a voter is defined by the majority of its input. If two or three inputs are equal, the output is set to that input value. If all of the three values are different, the output is not defined. For example if the component A3 fails, each of the voters V1 to V3 gets a false input value but the majority of the input values overrules A3 because the inputs from A1 and A2 are correct and equal. Therefore the faulty component A3 is masked and every component in the second stage (B1 to B3) gets a correct input value. This TMR-system even covers the fault of three different components, for example A1, B2, and C3. The voters them self are components and therefore they have to be redundant, too. For example if V1 fails the input of B1 is false but if the rest of the systems works correct, the voters V4 to V6 mask the fault in V1. Therefore a fault in V1 is the same as a fault in B1. The concept of *triple modular redundancy* is used in many safety-critical systems that are in productive use. For example the Austrian Federal Railways (ÖBB) use a system called *Elektra* for a fault-tolerant railway interlocking system [KK95].

The *Elektra* systems uses a second design paradigm which makes the system more fault tolerant which is called *N-version-programming* [CA78]. *N* stands for the number of diverse programs (or systems). For example imagine a system that consists of two individual subsystems. The first one is an Intel based standard PC, the second one is an IBM PowerPC, therefore the system uses two diverse hardware architectures. For the software components the developers can write the first control-software in *Assembler* and the other one in high level *C*, or whatever programming language fulfils their needs.

Fault removal is a concept which is important during the whole life cycle of a software component or a system. There two different terms which are used in the context of fault removal:

- *Verification* is used to guarantee that the system or software conforms to the requirements that are given in the *specification*. Therefore verification is used by system developers to check if they did the right job, and if the implementation of their work fulfils the specified behaviour. Verification is established at a low-level of system development.
- *Validation* is used to guarantee that the system fulfils its intended use for the purchaser. The question to ask is: "Does the product solve the problem?" Validation is a high-level concept which questions the whole system.

The verification of systems can be done by static or dynamic methods and the first step of verification is *testing*. If the testing process shows that the system violates its specification two further steps have to be undertaken. The next step is *diagnosing* why the tests have not been successful. When the developers know why the tests have failed, for example from the test results, they can make the final step which is the *correction*. The process of *testing*, *diagnosis*, and *correction* is an iterative one and has to be done during the whole life-cycle of a component. *Testing* is one of the most important activities during software development. The various methods of testing will be described in section 3.3.

Fault Forecasting is used to evaluate the system with respect to its faults [ALR01]. There are different methods which are used in real life which help system developers to examine potential faults. One of them is the *Fault Tree Analysis* (FTA) [Sta02], which is a graphical way to represent events that can lead to a fault. On top of the tree there is the fault and its sub-tree contains events that can lead to this fault. Sub-trees are joint with "AND" or "OR" operators (conventional logic gate symbols). A simple example is given in figure 3.7.

A second method of analysing the system according to potential faults is the *Failure Mode and Effect Analysis* (FMEA). The FMEA [Kop97] is done via a standardized work-sheet and helps to spot failures in the design process. A sample work-sheet can be found in table 3.2.

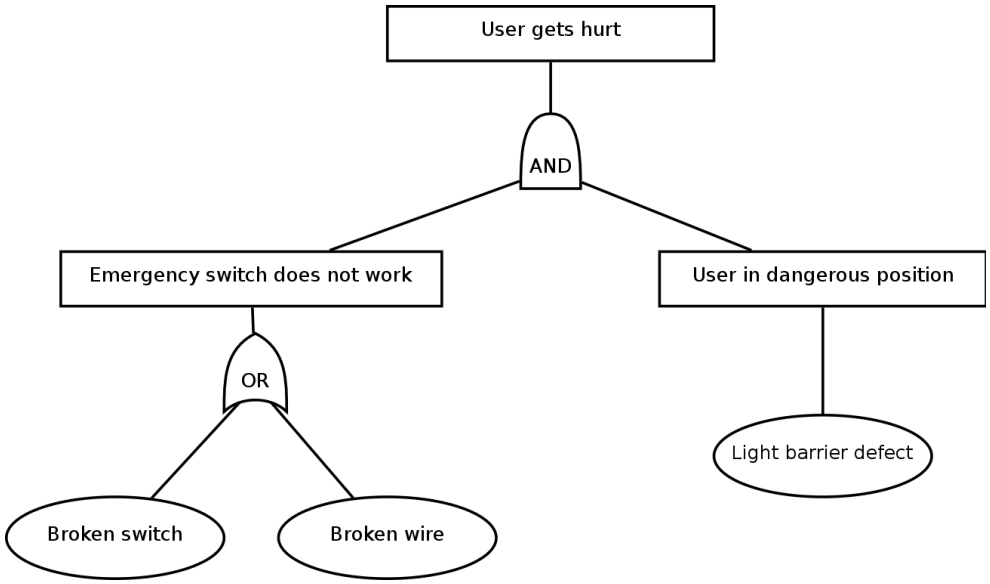

Figure 3.7: Fault tree analysis

Table 3.2: FMEA worksheet

Component	Failure Mode	Failure Effect	Probability	Criticality
Fan	Voltage drop	Overheating	1/10	high
Flash disk	Wear out	No further writes	3/10	high
...	...	...	...	...

3.3 Testing

Several safety-related standards force a certain level of testing and some of these standards even force certain methods of testing. Even if testing is not a formal requirement for a standard it is not possible to build a safety-critical system without testing under real life conditions.

The first two questions to ask are "What is testing?" and "Why should we do it?". Myers [Mye79] gives the following definition: "Testing is the process of executing a program or system with the intent of finding errors", whereas Hetzel [Het88] writes that "testing is any activity at evaluating an attribute or capability of a program or system and determining that it meets its required results". One famous quote by Dijkstra [Dah72] is that "testing can prove the presence of bugs but not their absence". While this is absolutely true, the process of testing is important because it leads to the following advantages:

- Improve quality
- Reliability estimation
- Verification and validation
- Certification

- Save costs (maintenance)
- Save lives (in a safety-critical context)

Summarized, testing increases the confidence that a system meets its specified behaviour. The process of testing is divided into two parts:

- Testing: Different kinds of tests are run and as a result someone knows if the system failed or met the specified behaviour.
- Diagnosis: This is the part where the interpretation of the results happen. Why did the test failed?

The following sections discuss the most important test methods.

3.3.1 Black-Box Testing

As the name of this testing method implies, the system (or the software) is treated as a "black box". It is an *external view* of the system and also known as functional testing [Kri02]. The most important characteristics of black-box testing are:

- Test cases are derived from the functional specification of the system.
- Only input and output interfaces of the system are considered.
- No implementation details are used to build test cases.

One advantage of this approach is that the generation of black-box test cases can be done at an early stage in the product life-cycle, because if the specification is known, someone can start to generate test cases. Black-box testing cannot exactly localise the cause of the failure (e.g. the source code line), it gives a general overview if the systems reacts to its specification.

It is impossible to test every existing scenario in test cases. For example, if the software accepts alphanumeric input strings up to a length of 100 characters this would lead to $\sum_{i=0}^{100} 62^i$ individual test cases. Therefore it is important to divide the test cases in *equivalence classes*. If it is expected that two different inputs cause an equivalent output they are in the same equivalence class. For example if the system under test is a simple calculator which adds two integer numbers, the input $2 + 3$ is equivalent to $4 + 2$ because the expected behaviour is the same. A test case that is not equivalent to the former test cases is the addition of a positive number and a negative number like $2 + (-1)$ because this tests if the calculator can handle signed numbers. A third equivalence class could be tests where the result is negative like $2 + (-3)$. A forth equivalence class could be tests where someone tests what happens if the range of an integer variable is exceeded, like $\$INT\_MAX + 1$. This class of test cases is the important class of *extremal values*. Additionally test cases are generated that examine the *normal case* (NC), *special cases* (SC), and *failure cases* (FC). An example for the former mentioned simple calculator is shown in table 3.3.

According to Kristiansen [Kri02] black-box testing has the following advantages:

- The test is unbiased, since the designer of the component and the tester are independent of each other.

Table 3.3: Example of black-box test cases and equivalence classes

Nr.:	Class	Description	Expected Result	Input
1	NC	2 positive integers	5	$2+3$
2	NC	2 negative integers	-8	$(-3)+(-5)$
3	NC	1 positive, one negative, result > 0	2	$4+(-2)$
4	NC	1 positive, one negative, result < 0	-1	$4+(-5)$
5	NC	1 negative, one positive, result > 0	3	$(-3)+6$
6	NC	1 negative, one positive, result < 0	-2	$(-3)+1$
7	SC	\$MAXINT + 1	warning message	$\$INT\_MAX+1$
8	EC	alphabetic character + integer	warning message	$a+3$
...	...	...	...	...

- The test case can be described as soon as the specification of the component is completed.
- The tester does not need knowledge of any specific programming language.

Therefore it is possible to outsource the testing from the skilled programmers to other persons that do not need to have a special insight in the software. Black-box testing can be done by persons that are able to understand the specification of the software component. Black-box testing is kind of "intelligent guessing", but design of the test cases and wise equivalence classes is needed.

There are several drawbacks [Kri02] of black-box testing, too:

- It is impossible to test every possible input.
- The test cases can be difficult to design, either because the component can take a wide variety of input, or because input can be combined in many ways.

3.3.2 White-Box Testing

In this form of testing the object under test is seen as a "white box", which means that the internal structure of the object has to be known. If someone wants to do white-box testing, he or she has to know the source code of the software under test. White-box testing needs an *internal view* of the system. White-box testing is defined by the following attributes:

- Generation of test cases is based on the knowledge of the program design or implementation.
- Verification of structural coverage requires white-box testing.

The goal of white-box testing is that every path or edge of the control flow diagram of a software component is executed. White-box testing cannot prove (in a mathematical sense) that the software is correct, but it can increase the possibility and the confidence.

There are several different kinds of coverage levels that are referred as C_x *coverage*, where "x" is the length of the edges in the software control flow diagram [Gre01].

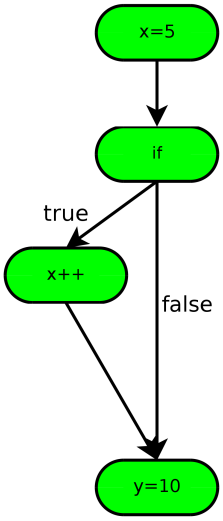

Figure 3.8: A valid C_0 coverage

C_0 coverage is the weakest coverage analysis and it is known as *vertex* or *instruction coverage* because it forces that every *statement* in the source code is executed at least once. This is demonstrated in figure 3.8. As we can see, it is sufficient to test every *statement* in the code.

C_1 coverage, which is also known as *edge coverage* has stricter requirements because it forces that every edge of the control flow diagram has to be executed at least once by a test case. This is shown in figure 3.9. Together the two test cases are a valid C_1 coverage.

To see the difference between C_0 and C_1 coverage, see figure 3.10. Every statement is covered by the test, but not every path. In this example the path that is executed if the if-statement evaluates to false is not checked, which violates C_1 coverage.

There exists a C_∞ coverage which is nearly impossible to reach in real life software projects. C_∞ coverage is also known as *path coverage* and forces that every possible sequence of statements is covered by a test case. In real life there are too much possible sequences that such test cases could be designed manually. One possibility to escape that dilemma is to use automated testing tools. Figure 3.11 shows two test cases that both together describe a valid C_1 coverage but do not guarantee a C_∞ coverage. The requirements from C_1 are satisfied because very edge is tested but not every *combination of paths* is tested. If someone wants C_∞ coverage he or she would need to specify the test cases in the following way:

- if == true, case a
- if == true, case b
- if == false, case a
- if == false, case b

Another form of coverage analysis is *condition* coverage which tests every possible combination *in* a conditional statement. For example, consider the following part of source code:

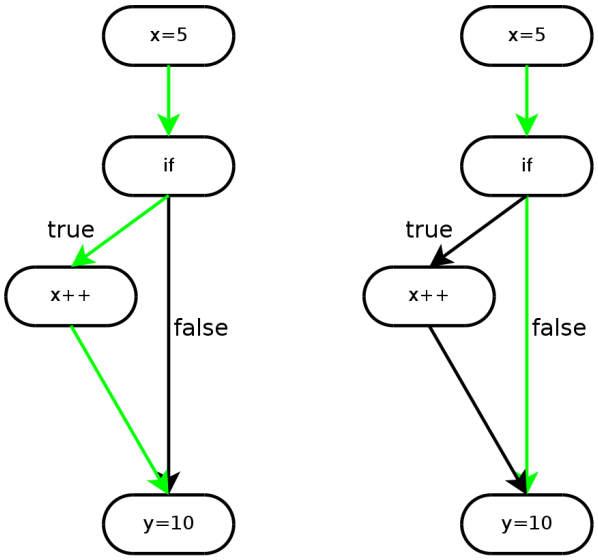

Figure 3.9: A valid C_1 coverage

`if((a == true) AND (b == false)) THEN ...` A successful C_1 coverage would need two test cases, one that results in *true* and one that results in *false*. For example, the following two test cases guarantee C_1 coverage:

- a == true, b == false ⇒ TRUE
- a == true, b == true ⇒ FALSE

For condition coverage someone needs these four test cases:

- a == true, b == true ⇒ FALSE
- a == true, b == false ⇒ TRUE
- a == false, b == true ⇒ FALSE
- a == false, b == false ⇒ FALSE

Summarized, white-box testing has the following advantages:

- If the source code is available, white-box testing can find programming errors much quicker than black-box testing. For example, vulnerabilities are found easier if someone reads the source code. If the program is a black-box it is much more difficult to find such vulnerabilities.
- With white-box testing it is easier to test specific code sections. It is possible to direct test the code. Black-box tests are more difficult to construct. Black-box testing makes it difficult to drive special input to an inner component of the program, whereas white-box testing supports this behaviour.

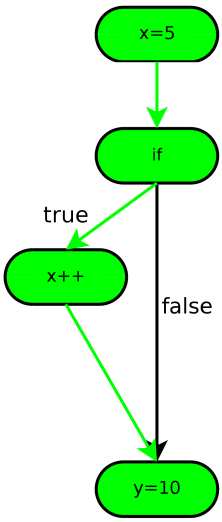

Figure 3.10: A valid C_0 coverage which is not a valid C_1 coverage

Apart from the former mentioned advantages, there exist several disadvantages:

- Source code is needed. White-box testing is only possible if someone has access to the underlying source code. This can be a problem in the context of proprietary software (e.g. proprietary COTS components).
- Complexity. The tester must understand the software in detail and the tester needs the ability to read and understand source code.
- If the source code changes the whole testing process needs to be redesigned and redone.
- White-box testing is more expensive than black-box testing. The know-how of the testers costs more and special tools that support this form of tests are needed. For example code compilers that support code coverage analysis.

In general white-box testing is superior to black-box testing but harder to do.

3.3.3 Fault Injection

Fault injection is done to observe the system under test under fault conditions and its reactions on these injected faults. The injection itself is done by software or hardware according to the system under test. Embedded systems are often tested via special hardware devices that are able to inject faults. These hardware injected faults can also have influence on the software of the device, for example the fault triggers a fault removal subsystem in the firmware of the device. The system under test gets two forms of input. The normal input data and the injected faults. Therefore fault injection can be seen as a special form of input data and the tester is interested in the output data of the system.

According to Kopetz [Kop97] fault injection has the following two purposes:

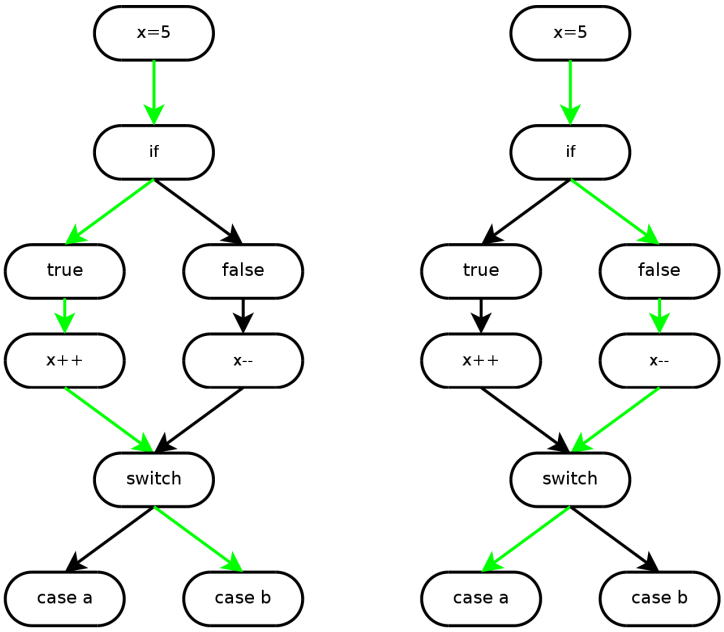

Figure 3.11: A valid C_1 coverage which is not a valid C_∞ coverage

- Testing and debugging: To test fault tolerance mechanisms someone has to generate artificial faults to trigger the tolerance mechanism. During normal operation faults are rare and therefore they have to be generated in order to guarantee that the system, and the fault tolerance mechanism in the system, has been sufficiently tested.
- Dependability forecasting: The data gained from the previous step can be used to make an assumption about the dependability of the system under test. This data is only significant if the operational environment of the system is known and if the tester provides a simulation of this environment that is sufficient realistic.

Table 3.4 gives an overview about the two different purposes of fault injection.

Table 3.4: Fault injection for testing and debugging versus dependability forecasting [AALC92]

	Testing and Debugging	Dependability Forecasting
Injected faults	Faults derived from the specified fault hypothesis.	Faults expected in the operational environment.
Input data	Selected input data to activate the injected faults.	Input data taken from the operational environment.
Results	Information about the operation and effectiveness of the fault tolerance mechanism.	Information about the envisioned dependability of the fault-tolerant system.

Fault injection can be done by two different approaches. The first one is *physical fault injection* and the second one is *software fault injection*.

Physical fault injection is mostly used to direct test hardware systems. One advantage of physical fault injection is that it can provide a more suitable simulation of the systems intended usage. For example if the system is a satellite, a bombardment with α-particles is a good approach to simulate the physical environment in which the satellite will be used. Kopetz [Kop97] shows three different methods of physical fault injection. The first one is a bombardment with α particles, the second one is pin level fault injection where potentials are forced directly on the hardware pins and the third one is electro magnetic interference (EMI), which is a form of radiation. According to Kopetz [Kop97] these methods have the characteristics shown in table 3.5.

Table 3.5: Different properties of physical fault injection [AALC92]

Technique	Heavy ion	Pin level	EMI
Controllability, space	low	high	low
Controllability, time	none	high/medium	low
Flexibility	low	medium	high
Reproducibility	medium	high	low
Physical reachability	high	medium	medium
Timing measurement	medium	high	low

The second form of fault injection is software based where the memory of a program is manipulated in several ways. The injection of these faults can be done by the system (software) under test itself or by special tools developed for fault injection.

There are several advantages of software based fault injection over physical fault injection [Kop97]:

- Predictability: In opposite to physical fault injection the instant of time and space is more predictable than with physical fault injection. For example the exact time and location where a fault is injected with methods of electromagnetic interference or with α-particle bombardment is hard to control.
- Reachability: Pin level fault injection has the limitation that several inner portions of the system cannot be reached because they do not have dedicated pins. With software based approaches every location of the memory or the registers can be changed.
- Effort: Hardware based fault injection needs special equipment which is extremely expensive and is mostly found at universities or rich equipped laboratories. Software based approaches are cheaper and much easier to handle.

A typical approach for injecting faults in the software itself are pre-processor statements that are activated if the software is compiled with this `#define` enabled. Following snippet of source code demonstrates this method.

```
int sum(int a, int b)
{
#ifdef INJECTFAULT
    return(23);
#else
    return(a + b);
```

```
#endif
}
```

This form of fault injection has the advantage that is can be done very easy. On the other hand it is not very flexible and some of the modern programming languages do not have the support of pre-processor statements. For example *Java* does not use a pre-processor. If there is no pre-processor the former mentioned code can be rewritten the following way:

```
int sum(int a, int b)
{
    int injection = 0; /* set to '1' to inject faults */
    if(injection)
        return(23);
    else
        return(a + b);
}
```

This has the disadvantage that it alters the control flow of the software which makes it harder to test and to verify a piece of software, because strictly the software with these if-else constructs is not the same as without them.

The second category of software implemented fault injection can be divided into two sub-categories. The first one are tools that are separate from the software under test but interfere with the software under test and change its behaviour. An example for such a tool is a *debugger* which is able to change the value of variables in the software under test. The second category are *virtual machines* that encapsulate the software under test and inject faults from outside. Some virtual machines provide the ability of fault injection, some use external tools like debuggers. The advantage of virtual machines is that it is easier to test core software components like the kernel of an operating system. The advantages of the three different approaches is shown in table 3.6.

Table 3.6: Different properties of software implemented fault injection

Technique	Directly in software	Debugger	Virtual machines
Complexity	low	low/medium	high
Flexibility	medium	high	high
Timing measurement	high	medium	low

3.4 Safety-Related Standards

Different standards force different requirements for safety-critical systems. The field of application is the railway domain which, is an appropriate example for an area of applications that force safety. Applications in this domain have the advantage of a *safe state*. The term will be explained in section 3.4.1. The standards concentrated on in this section are European standards that are approved by *CENELEC*, which is the "European Committee for Electrotechnical Standardization". The most important ones in this context are EN 50126 - "Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety

(RAMS)" [CEN99], EN 50128 - "Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems" [CEN00], and EN 50129 - "Railway applications - Safety related electric systems for signalling" [CEN03].

Standards in general do not provide a step by step procedure for certifying components according to them. For example, terms like "Evidence of quality management" are used, but the standard does not enforce a specific method how this evidence should be given. In general, this makes it harder for a system designer to choose an adequate approach, but it gives designers the chance to provide alternative resources of evidence, which opens the door to non-traditional design paradigms like the Open Source development model. Therefore, it will be examine if it is possible to provide sources of evidence, that the Open Source development model provides methods that make it possible to certify Open Source components according to the former mentioned safety-related standards.

3.4.1 Safe State

In the context of safety-critical systems and applications, it is important to identify the so called *safe state*, if it exists. In a safe state the controlled object is in a state where it cannot harm its users or the environment. For a railway signalling system, this safe state could be to stop all the trains, switch the signals to red, and do not switch any track switches. If this behaviour can be reached in a reasonable time, the system is called *fail-safe*. In practice these kind of systems provide a high degree of error-detection to be able to reach the safe state. There exist special devices which help to monitor the state of the system. For example, a *watchdog* is such a device, where the system under control has to send a life-signal in a fixed period of time. If the system does not send this signal, the watchdog assumes that the system has failed and forces the necessary steps to bring the system to a safe state. It is important to note that reaching the safe state is desirable from a safety standpoint, but it is a system state that is undesirable from a availability prospective. A simple safe state is shown in figure 3.12. Even if the cable breaks, the system is in a safe state, because the system gets turned off.

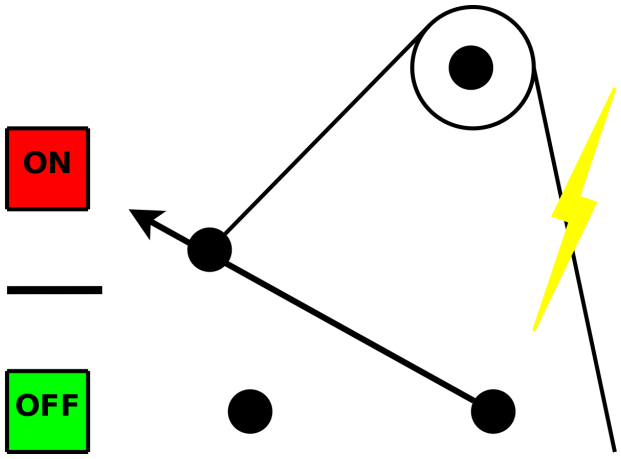

Figure 3.12: A simple example for a safe state

In contrast to railway systems, where a safe state can be identified (with the exception of a failure in a tunnel where the tunnel itself influences the overall safety), there exist systems where this

safe state does not exist. For example if the fly-by-wire system in an (already flying) aircraft fails, it does not make sense to stop the system. These kind of systems have to provide a minimum of operation to guarantee safety to its users, which is the reason why they are called *fail-operational*.

3.4.2 Requirements in Different Safety-Related Standards

According to EN 50129 [CEN03] there are three requirements that shall be satisfied for safety-critical systems:

- Evidence of quality management
- Evidence of safety management
- Evidence of functional and technical safety

The evidence that these requirements are satisfied should be provided in a *Safety Case*. It is not the aim of this thesis to provide such a Safety Case, it will be determined if the Linux community, and in general the Open Source community provides methods that can satisfy the strict requirements of safety-related standards. A discussion can be found in section 3.4.2.1.

Appendix A of the standard EN 50128 [CEN00] contains different tables that provide information how important different requirements are for different *Safety Integrity Levels*. There are five different types of importance which are summarized in table 3.7, which can be found in the appendix of the corresponding standard. A discussion of the mandatory requirements can be found in section 3.4.2.2.

Table 3.7: Different types of importance for different safety integrity levels [CEN00]

Importance	Description
M	This symbol means that the use of a technique is mandatory.
HR	This symbol means that the technique or measure is Highly Recommended for this safety integrity level. If this technique or measure is not used then the rationale behind not using it should be detailed in the Software Quality Assurance Plan.
R	This symbol means that the technique or measure is Recommended for this safety integrity level. This is a lower level of recommendation than “HR” and such techniques can be combined from part of a package.
-	This symbol means that the technique or measure has no recommendation for or against being used.
NR	This symbol means that the technique or measure is positively Not Recommended for this safety integrity level. If this technique or measure is used then the rationale behind using it should be detailed in the Software Assurance Plan.

3.4.2.1 EN 50129 Evidence of Quality Management

EN 50129 [CEN03] provides a list of example aspects that should be considered for appropriate quality management for safety-critical systems. The following list is a selection of these aspects with a short explanation how the Linux community satisfies this requirement (with a reference to the relevant chapter in this thesis).

- Organisational structure: The structure of the Linux development team is hierarchic and there is a small group of high qualified decision makers (Section 2.4.2 and figure 2.9).
- Quality planning and procedures: The code quality is ensured by the development model itself. There are multiple levels of reviews until code is included in an official released kernel (section 2.4.2).
- Handling and storage: As shown in section 2.4.3, the Linux (and Open Source) community has developed professional tools that make it possible to handle the kernel code in a comfortable way. In addition, the code handling system (git) guarantees code integrity, which is important in the context of safe and secure systems.
- Inspection and testing: There are different levels of inspection and testing in the Open Source development model. First of all Linux kernel code has to pass multiple levels of authorities where the code gets inspected by different maintainers. Remember the persons in charge for a file (file maintainers) or for a sub-architecture (sub-architecture maintainers) from section 2.4.2. On the other hand there exist several dedicated projects for kernel level testing. For example the *Linux Test Project* (LTP), which was presented in section 2.5.
- Non-conformance and corrective action: Linux and most GNU/Linux distributions provide sophisticated methods to deal with this mentioned "corrective actions". As shown in section 2.4.4, there are different branches of the kernel itself. A system designer has the chance to choose a stable tree of code. If there are bugs, the designer does not have to choose a newer version because that could violate the compatibility, he or she can stay in the same tree and gets fixes for it. These are the kernel releases with four digits (a.b.c.d), named the *stable tree*.
- Quality monitoring and feedback: By the open source nature of the kernel, developers get much feedback from their users. There are different ways how a user can give feedback to the developers. General discussion can be done on the kernel mailing-lists [28]. Bug submission can be done via mailing-lists, too, but there are at least two different ways. The suggested way [20] is to contact the specific developer that is involved with this issue. As seen in section 2.4.2, there is a maintainer for every single file. The third way of submitting bugs is to use *Bugzilla* [4] [43]. Bugzilla is a tool that helps managing software development. It is installed on a web-server and provides a simple web interface for submitting bugs. Bugzilla has shown that it is suitable to support huge software projects and it is used by various projects and companies like Mozilla, Gnome, KDE, Apache, Open Office, Red Hat, Novell, and the NASA.
- Documentation and records: The kernel package includes a set of documents which describe the functionality of individual kernel parts. The documentation can be found in a directory called "Documentation" in the top-level directory of the kernel sources. At the moment the documentation contains over 900 individual documents.

The two other requirements ("Evidence of safety management" and "Evidence of functional and technical safety") are dependent on the purpose of the safety-critical system. These requirements cover the system or sub-system as a whole and it does not really make sense to pick one component, the kernel or even the whole operating system, because the overall safety is influenced by much more components. The overall safety includes hardware, too. For example a less safe software component can be tolerated if adequate design decisions have been kept in mind, like TMR or N-version programming.

3.4.2.2 EN 50128 Mandatory Requirements for SIL4 Applications

The standard *EN 50128* provides a list of requirements that are necessary for SIL4 systems. A discussion which of them are fulfilled by the kernel Linux follows. Refer to table 3.8 for a list of mandatory requirements and their aim according to appendix B of the corresponding standard.

Table 3.8: Mandatory requirements for SIL4 applications [CEN00]

Mandatory Requirements	Aim or Requirement
Modular Approach	Decomposition of a software system into small comprehensible parts in order to limit the complexity of the system.
Design and Coding Standards	To ensure a uniform layout of the design documents and the produced code, enforce egoless programming and to enforce a standard design method
Functional/Black-box Testing	No further comments in appendix B [CEN00]
Performance Testing	A suitable set of techniques shall be chosen according to the software safety integrity level.
Data Recording and Analysis	To record all data, decisions and rationale in the software project to allow for easier verification, validation, assessment and maintenance.
Compliant with EN ISO 9000-3	Self explaining
Company Quality System	No further comments in appendix B [CEN00]
Software Configuration Management	Software Configuration management aims to ensure the consistency of groups of development deliverables as those deliverables change.
Impact Analysis	To identify the effect that a change or an enhancement to a software system will have to other modules in that software system as well as to other systems.

Before examining the requirements in detail, it should be mentioned that it is not enough, if these strict requirements are fulfilled only by the kernel, the system is much more than a kernel. Even if some aspects of the requirements are not fulfilled by the kernel at all, the system itself can be safe if other adequate methods like diverse programming or fault tolerance are applied. It is out of the focus to provide these strategies for a whole safety-critical system, therefore it is only examined how Linux, the kernel, applies to the requirements of the standard.

The approach that is taken for Linux development is in deed modular and fulfils the first requirement. As mentioned in chapter 2 the kernel is developed in an open and distributed way, which is the nature of Open Source. It simply makes sense to break the challenge of kernel development into smaller challenges. Remember that the development is organized in a hierarchic way. There are encapsulated classes of sub-architectures like USB, Firewire, Networking, and so on. That makes it much easier for developers - even if they are new to kernel development - to concentrate on their special field of interest. For example, someone that wants to improve the networking stack, does not have to be familiar with the Firewire stuff. Another indication for the *modular approach* is the concept of *kernel modules*. If it is possible to deactivate code or even whole subsystems and activate them at runtime, it is a indication of a working modular approach.

Design and coding standards are defined by kernel developers, too. There is a document in the kernel tree (`/usr/src/linux/Documentation/CodingStyle`) that describes the preferred coding style, which is different from the GNU coding standards. The following list contains a few selected rules from the former mentioned document.

- Tabs are 8 characters wide.
- Indentation of 8 characters.
- Multiple statements should not be on the same line.
- Statements longer than 80 columns should be broken into smaller parts.
- Opening braces should be put on the same line as the statement (Kerninghan and Ritchie style).
- Enforcement of short but descriptive variable names.
- Functions should be short and do one thing. Typical size of two 80x24 screens. If they are complex, it is okay if they are longer.
- Local variable count of about 5 to 10.
- Comments should explain what a function does and why, but not how. This enforces that the function code is easy to read and easy to understand.

Kernel-level testing projects have been presented and discussed in section 2.5. The LTP provides a modular and extensible framework for black-box testing. In addition code coverage analysis can be done with the tools from the LTP. The *autotest* project provides sophisticated methods for the second requirement, namely *functional testing*. The results can be found on `http://test.kernel.org`.

There are two major projects for *performance testing*. The first one is the former mentioned *autotest* project, the second one is the *kernel-perf* project.

Data recoding and analysis is given because all kernel data, in the sense of source code, is recorded on the kernel homepage [27]. Even the source code of the first kernel release (version 0.01) is available on this project page. For newer kernel releases someone can use *git*, which was described in section 2.4.3. Git makes it easy to check out a specific tree or version of the kernel tree. Besides from that, the history is logged in the git commit-logs.

Compliance to *EN ISO 9000-3* and the *company quality system* are requirements that are out of focus here. The question if the company quality management is sufficient can not be answered in a general manner, because it depends on the specific safety-critical system and on the specific company and its quality system.

According to appendix B of the standard, *Software Configuration Management* means the recording of the production of every version of every "significant" deliverable [CEN00]. For example such a significant component is a kernel module or a kernel sub-system. The whole life-cycle of such a component, in principle of every source code file, is recorded in its *git* history. It is easy to compare two different versions of such a source code file. Additionally the commit log files (`man git-log`) provide information which changes happened during the component's life-cycle.

Concerning the *impact analysis* and its consequences as described in appendix B of the standard, the history of Linux has shown that the kernel community can successfully deal with this dynamic process. In this context the *verification* is done by the kernel developers them self and in addition by the former mentioned testing tools. Appendix B of the standard describes the following decisions:

- Only the changed module to be reverified: This is the most common case, because the kernel module is encapsulated in a sub-architectures. Therefore the usual case is that a change in one component does not effect a different sub-architecture. Of course, there are interconnections that are sometimes difficult to identify, but during the history of Linux, especially from version 2.4 to version 2.6, things changed to the better.
- All identified affected modules are reverified: There have been major changes in the kernel architecture since the beginning of the kernel and they continue until now. For example the integration of SMP forced major changes in the architecture, but history has shown, that they were successful and therefore, that the way the kernel is developed is a healthy one. The kernel gets verified by automatic testing tools, too.
- The complete system is reverified: Except from the early days there never was a situation where the whole kernel has been rewritten from scratch. This would not be possible nowadays, because the kernel reached a enormous size and complexity. On the other hand, the kernel is reverified as a whole while it is developed by the tools mentioned before, by the developers, and by the users.

3.5 COTS

Section 3.5.1 gives a definition of the term COTS and examines their advantages. Section 3.5.2 analyses the existence of COTS in the context of GNU/Linux.

3.5.1 Definition of COTS

Nowadays developing software is a complex matter and writing whole operating systems from scratch is impossible or it would lead to enormous costs. Therefore system designers try to reuse components that are already tested, verified or even certified. This reused components (hardware or software) are referred as COTS components. In the literature, the abbreviation COTS stands for *Commercial Off The Shelf* or sometimes for *Component Off The Shelf*. An abbreviation that

should be mentioned in this context is *SOUP* which stands for *Software Of Unknown Pedigree*. In the context of this thesis COTS is used as *component* off the shelf because the reuse of components does not have to be a commercial one. There are several definitions what COTS means and which attributes it has, like the following by Morisio and Torchiano [MT01]:

- Exists a priori
- Is available to the general public
- Can be bought, or leased or licensed
- The buyer has no access to the source code.
- The vendor controls its development.
- It has a non-trivial installed base.

Jones [JBFB01] defines the following characteristics of SOUP:

- SOUP already exists and cannot be re-engineered.
- It is generic and hence may contain functions that are unnecessary for the system application.
- It is subject to continuous change. A mass market SOUP will evolve to meet customer demands and to match the competition.

These assumptions lead to the following definition of COTS software for Free and Open Source Software (FOSS):

- It exists a priori and can be re-engineered.
- It is available to the general public.
- The buyer has access to the source code.
- It is generic and hence may contain functions that are unnecessary for the system application.
- is subject to continuous change.

The kernel Linux fulfils every point of this definition. On the other hand there are distributors of GNU/Linux systems that do not fulfil all of them. For example some of them provide third party tools which are not released as FOSS.

According to the paper "COTS, Integration and critical Systems" [Sta97], COTS components have the following advantages:

- Reduces software development costs
- Reduced software maintenance costs

- Improved reuse

As it is known, developing software is an error prone process and therefore it does not make sense to reinvent the wheel over and over again. This reinvention can be avoided if COTS components can be used. This saves time and therefore development costs. A well defined COTS component can be seen as a black-box with well defined interfaces and maintenance costs can be saved, because the system designer has well defined interfaces to the component but the inner structure can be hidden. An example for this is a software library which provides interfaces in the form of class definitions. If there is a bug in the class, the developers can fix them, or they can improve the performance of the class by implementing new algorithms but the external interface, the class definition is still the same. If the component, software as well as hardware, is well designed it can be used for different projects and this improves the reuse of COTS components.

On the other hand COTS components can have several disadvantages [DK97]:

- The potential for introducing systematic errors through component mismatch
- Meeting the obligations of the system safety case

Developing customized software is a process of specifying, implementing, and reviewing the system under development. There is always some loss of information between the specification and the resulting system. Sometimes the COTS component cannot be seen as a black-box, it has to be seen in the context of the whole system. Uncoupling such a component can lead to several problems. One famous example is "Test flight 501" from the *Ariane 5* rocket in 1996. The control software, which was successfully used in the *Ariane 4* program, was reused but had a malfunction in the context of the new *Ariane 5* system which lead to a self-destruction after 37 seconds airtime [Lio96]. If a component is reused, the view of the system as a whole is lost, and therefore the COTS-component has to be reviewed if it successfully fulfils the properties of the new system.

The claims that a component is safe in the context of its safety case is based on the argument that it is "sufficient tested and verified". But this argument is valid in the context of the whole system and does not have to be valid in the context of the new system where a component is reused.

3.5.2 GNU/Linux Distributors and Their COTS Products

Several well known GNU/Linux distributors and their range of products are examined in this section. The focus is on the attributes that are most important in the context of this thesis. These criteria include the level of support from the distributor as well as the long term support of the distribution. As seen in the previous chapters it is tedious to certify products and to design systems that meet the high requirements for safety-critical systems. Therefore most safety-critical systems have a relatively long operation period. Unlike the traditional consumer market, safety-critical systems have decades of operation. It is difficult to find vendors that provide this long support periods and if someone chooses such a vendor or product he or she should keep that in mind. If the vendor does not offer this level of support, it is an important decision to take this risk [Voa98]. "The Cost of COTS" [Tal98] describes an interesting way of solving this particular problem. The customer can make an agreement with the vendor that at the end of the support

period, the customer gains access to the source code and can therefore support the product himself. In addition to that, a special focus on products that are sufficient for safety-critical systems will be set. For example, this will include the availability of products that have special real-time features.

3.5.2.1 Red Hat

Red Hat offers a huge quantity of desktop and server products. The most important product in this context is the "Red Hat Enterprise 5 Linux Server" which is available in a basic version and a so called "Advanced Platform" version. The most notable enhancement is its higher level of virtualisation. The advantages in the field of safety that may be gained through virtualisation will not be examined here, but could be determining for the choice of the GNU/Linux distribution.

Red Hat offers different kinds of support-levels through a *subscription*. According to Red Hat [37] they offer a duration of support for 7 years [38].

The general support includes:

- Product updates & upgrades
- Covered under the Open Source Assurance program
- Supported by leading ISV (Independent Software Vendor) applications
- Certified on leading OEM (Original Equipment Manufacturer) hardware
- Includes full suite of open source server applications, including: Apache, Samba, nfs, ftp, Tomcat, MySQL, PostgreSQL and network servers
- Installation and documentation media kit - DVDs only (optional purchase)
- Does not include desktop applications

Depending on the chosen server version, Red Hat offers the following support:

- Web and phone-based comprehensive support, 24x7 coverage, 1 hour critical incident response (4 hour normal), unlimited incidents, Red Hat Network Update.
- Web and phone-based comprehensive support, 5x12 coverage, 1 hour critical incident response (4 hour normal), unlimited incidents, Red Hat Network Update.
- Web-based comprehensive support, 2 business day response, unlimited incidents, Red Hat Network Update.

Security issues are treated by a *Security Response Team*, that investigates and verifies the issue. After that, the team analyses which products are infected and what impact it causes on these products. If a security update has to be done the team will create one which has a minimal side-effect to the other services.

The update and management of a Red Hat system is done via a so called *Network Update Module* which offers the following features [41]:

- Simple web user interface
- Priority email notification
- Errata information
- RPM dependency checking

The former mentioned support for 7 years is divided in three different periods. The first phase is *full support* for 3 years, followed by a 3.5 year *deployment support*, and a 7 years *maintenance support*. According to the Red Hat Enterprise Linux Errata Support homepage [39], the policy is structured way table 3.9 presents.

Table 3.9: Red Hat's errata policy

Errata Policy	Start Date	End Date	Description
Full Support	General Availability	3 years from general availability	During the Full Support phase, new hardware support will be provided at the discretion of Red Hat via updates. Additionally, all available and qualified errata will be applied to the enterprise products via updates (or as required for security level errata). And finally, updated ISO images will only be provided during phase 1: Full Support.
Deployment Support	3 years from general availability (end of full support)	3.5 years from general availability	During the Deployment phase, all available and qualified security and bug fix errata will be applied to the enterprise products via updates. Security errata will be released as necessary independent an update.
Maintenance Support	3.5 years from general availability (end of deployment support)	7 years from general availability	Description: During the Maintenance phase, only security errata and select mission critical bug fixes will be released for the enterprise products.

In December of 2007 Red Hat made a formal product announcement of a product that supports some kinds of real-time extensions. Red Hat was one force of Linux real-time enhancement for several years. This product is called *Red Hat MRG* (Messaging, Real Time Grid) platform. The core component is a real-time enhanced kernel that replaces the normal kernel of the Red Hat Enterprise Linux product. If a company runs Red Hat certified products it is no problem to run them on Red Hat MRG.

According to the product page of MRG [40], it provides these enhancements:

- A real-time kernel, which replaces the Red Hat Enterprise Linux 5 Kernel

- A set of configuration utilities designed to allow real-time tunign.
- A set of performance monitoring tools to allow real-time performance and be monitored and provide information to support real-time tuning.
- Documentation, including information on how to tune a system for optimal latency.

The real-time enhancements are only in the kernel. The rest, like runtime environment, system utilities, and glibc, is the same as in the Red Hat Enterprise Linux 5. The update and fix cycles are the same as for Red Hat Enterprise Linux, too.

3.5.2.2 SUSE Linux

SUSE Linux [48] owned by *Novell* is one of the big players in the distribution business and offers GNU/Linux distributions since the early nineties. Like Red Hat, the SUSE distribution uses *rpm* for its package management. The most interesting product in the field of COTS is the *SUSE Linux Enterprise Server* (SLES).

The administration of a SLES system is done by a set of tools that were designed by SUSE and Novell special for the purpose of system administration. These tools are called *YaST* and *AutoYaST*. YaST provides a graphical user interface (GUI) and a commandline interface for remote administration. According to Novell YaST can be used for system installation, hardware setup, network configuration, software and packages selection, and service setup. It is used to configure and manage every aspect of the server. AutoYaST extends the concept of YaST in the field of administering multiple servers. For example software can be installed on a range of different servers in a simple way.

Novell offers a range of different support levels for their products. These range from *standard and priority subscriptions* to *premium* services. The standard and priority support offers the level of support shown in table 3.10.

Table 3.10: Standard and priority support for SUSE Linux

Property	Standard	Priority
Access	12x5	24x7
Maximum Target Response Time	4 hours	1 hour
Service Requests	Unlimited	Unlimited
SUSE Linux Enterprise Server Fundamentals Training (Course 3071)	Included	Included

Novell is the second GNU/Linux distributor examined in this thesis that offers a product with the label “real-time” for time critical servers, which was even announced before Red Hat’s MRG. The product is called *SUSE Linux Enterprise Real Time* [49] and according to the Novell’s web page it is deterministic, has low latency, and runs on a variety of industry standard hardware platforms. The description of the product includes safety-critical systems like “industrial control” or “medical imaging systems”. Novell writes that “properly configured, SUSE Linux Enterprise Real Time ensures that a user-level application can respond to an external event in less than 30

microseconds" [50]. Further Novell writes: "It is capable of running on a variety of industry-standard hardware. Ideally, SMP machines (x86 of x86/64 architectures) run with at least one dual-core CPU. Everything that is required for a real-time solution is included in the box. That includes kernel modifications for real-time supporting user libraries, low latency node interconnect infrastructure (Infiniband), additional documentation, and command-line performance and monitoring tools". According to the product's web page the following main changes to a traditional GNU/Linux system have been undertaken:

- CPU assignment and shielding: Processes that have special real time characteristics can be assigned to a dedicated CPU or CPU-core. Shielding prevents non real time tasks from interfering with real time tasks on such a dedicated CPU.
- SoftIRQ enhancements: If a devices raises an interrupt it is split into two parts. The first part is handled at interrupt priority, which is always higher than the normal priority of the process, and handles only the most critical aspects of interrupt-completion processing. The second and not that critical part is deferred to ran at program level. This strategy makes sense because the system can achieve better program scheduling latency by removing non-critical processing from interrupt level. The second part of the interrupt routine is handled by kernel deamons. Novell's kernel patches allow tuning of these kernel deamons by the administrator of the system. Therefore the administrator can set the priority of the kernel deamons that fulfil the processing of the second part of the interrupt handling, which is de deferred part. Therefore the administrator can set the priority to set the priority of the deferred interrupt kernel deamon so that a high priority user process overrules the priority of the deferred interrupt kernel deamon, which delivers a more deterministic response time for the real time tasks that execute on this CPU.
- Priority inheritance: This is a modification to the kernel's internal semaphore/mutex mechanism. In a traditional system a lower priority task can block a high priority task by not releasing a resource (e.g.: memory, hardware resource), that the lower priority process already has under control. With the patches provided the former low priority process inherits the high priority of the process that wants to get the resource until it releases it. Therefore time is saved because the process that has now a higher priority will be executed in favour of other lower priority processes. SUSE Enterprise Real Time includes a modified version of the "glibc" (GNU C library) that extends the concept of *priority inheritance* to the user space.
- High precision timers and process accounting: The POSIX kernel timer service has a resolution of about 40 milliseconds and it was replaced by a high precision timer that has a resolution in nanosecond interval.
- Low latency connections: The product uses Infiniband [21] as a low latency node connection which improves network connectivity and throughput.

3.5.2.3 Ubuntu Server

Ubuntu [53] is a relatively new GNU/Linux distribution which is based on the well known *Debian GNU/Linux* distribution [6]. Debian GNU/Linux itself is one of the oldest Linux distributions and known for its stable base systems. Therefore, it is one of the most used operating systems

for mission critical server systems. Ubuntu is known for its user-friendly desktop distribution but provides a special flavour with the focus on server systems.

The Ubuntu community provides these different distributions (see table 3.11 for details).

Table 3.11: Ubuntu GNU/Linux products

Ubuntu Flavour	Version	Support
Desktop	Ubuntu 8.04 LTS	Supported to 2011
Server	Ubuntu 8.04 LTS	Supported to 2013

The version number contains the release date of the distribution in the numbering scheme *YEAR.MONTH*. Therefore Ubuntu 8.04 was released in April (04) in the year 2008 (8). The term *LTS* means *Long Term Support* which is provided for 3 years for the desktop version and 5 yeas for the server edition.

Ubuntu is a community based distribution and updates and security fixes are provided by the Ubuntu development team. As known from Debian, Ubuntu uses *apt-tools* to manage software. *Apt* reads a special file (`/etc/apt/sources.list`) and according to its settings, it creates a new database of available packages. For example it is possible to stay at the stable packages which were provided during the release and only make updates to these packages if there are security fixes. For details examine the documentation of apt or the corresponding man-pages. Updates can then be applied in a simple way:

```
# resynchronize the package index
$ apt-get update
# install the newest version / security fixes
$ apt-get upgrade
```

As mentioned before, Ubuntu is a community based project, but it features a high professional hierarchy of developers in several different structures. The communication is done in traditional ways, like IRC-channels or mailing-lists. On the other hand Canonical Ltd. created a new tool which is called *launchpad* [24] that provides a platform for communication and development of free software. For example it is possible to submit and trace bugs or work and discuss important issues in a software project (e.g. the release cycle of a product).

The structure of the development team and there rights is shown in table 3.12.

This is only a list of the most important rights the corresponding developers have. For a complete list, please take a look at the Ubuntu wiki-page [35].

Canonical offers a commercial support for Ubuntu Server and Desktop systems. The support is provided via phone, e-mail or web for a duration of one year. There are two different options, one 9x5 support and a 24x7 support. Both include *live phone support*, *e-mail support*, *free updates*, and *security updates*.

Table 3.12: Ubuntu developers

Ubuntu Developers	What do they do? Which rights do they have?
Prospective Developers	Starting point for new developers. Work with an developer or core developer as sponsor and mentor. Give reviews and feedback. Interact with MOTU (member of the ubuntu-dev team in Launchpad). Can become developers.
Ubuntu Developers (MOTU)	Members of the *ubuntu-dev* team in Launchpad. Upload and updating of packages. Contribution to the core developers. Provide guidance for prospective developers. Participate in technical discussions.
Ubuntu Core Developers	Members of the *ubuntu-core-dev* team in Launchpad. Strong knowledge of the Ubuntu project procedures, containing releases and support commitments. Have a history of substantial contributions to the distribution. Sense of personal responsibility for the quality of the distribution.

3.5.2.4 Gobuntu

As it is discussed in chapter 4, it is a benefit for developers of COTS systems if the underlying components are free software. This makes it easier to certify a product, because the developers have access to the source code and can alter it according to their needs. Therefore it is important to examine if there is a GNU/Linux distribution that strictly uses free software. For example, many of of the commercial distributions include proprietary firmware or proprietary drivers. From a safety standpoint it is difficult to argue that these third party software conforms to the specified safety assumptions.

There exists a relatively new project which is *Gobuntu* [16] and can be used as a base for the development of COTS components. According to the project it follows the definition of the *four freedoms* introduced by the *Freesoftware Foundation* (refer to section 2.1 for details). The project is related to the *Ubuntu* project and uses the same concepts like *apt-tools* for package management. The Gobuntu wiki [17] mentions the following advantages of its distribution, where the last one is the most interesting for safety-critical systems:

- Someone likes Ubuntu technically, but is concerned about his/her freedom.
- Someone is a system administrator at an organization, which has a commitment to use only free software.
- Someone runs a system with high security requirements, allowing only software with source code available to be used.
- Someone is choosing parts for a commercial hardware platform. He/She wants to make sure that his company has access to all necessary bits of code so they are able to support their product in future.

In this case the last statement should be extended that it is easier for this platform to get certified.

Recent news show that the future of Gobuntu is uncertain because there is too less interest in the community. Fortunately the standard Ubuntu distribution offers a “Free Software only” installation nowadays.

4 Related Work

There are several papers that deal with Linux and COTS for safety-critical systems. The focus is laid on a discussion of papers that are relevant in the context of GNU/Linux, safety, and COTS. The first two sections deal with individual papers (section 4.1 and 4.2), whereas section 4.3 concentrates on specific topics relevant in the context of this thesis.

4.1 Preliminary Assessment of Linux for Safety-Related Systems

The paper "Preliminary assessment of Linux for safety related systems" from Ron Pierce [Pie02] is about the possible use of GNU/Linux for safety-critical systems. To sum up, he concluded that Linux may be feasible for SIL3 applications and cannot be used for SIL4 ones. This paper was written in the year 2002 and Linux has undergone some important changes which were not known at this time when Linux 2.4 was up to date. For example, this includes enhanced real-time capabilities and a notable different development and quality management model introduced during kernel 2.6 (see chapter 2 and the following discussion). Ron Pierce came to his conclusions by examining the potential of Linux from a vanilla[1] Linux (kernel) in version 2.4. Kernel version 2.6 brought a lot of extensions and new features that have been already discussed in the previous chapters, or will be discussed later in this chapter when a specific lack of a feature is claimed. The improvements of non-vanilla kernel projects like RT-Linux should have influence on the conclusions, too.

On page 11 Ron Pierce writes: "There is, as far as this study has been able to determine, no single definitive work which describes the services provided by Linux [...]". This is not true because Linux conforms to the POSIX specification and therefore the services that Linux (the kernel) provides are specified. On the same page Ron Pierce writes about POSIX that "it is regarded as the lowest common denominator of Unix systems and does not provide some commonly used functions such as the communications socket concept which is the means of providing network communication protocols (in particular, the Internet protocol IP and higher layers such as TCP and UDP)". This cannot be seen as a counterpoint for Linux, because if the safety-critical application does not need these services, there exists no problem. Imagine an embedded system that does not need socket communication. The whole subsystem can be disabled from the kernel at the kernel configuration step. This has the advantage that there does

[1] The term vanilla refers to that kernel that can be downloaded from www.kernel.org without any special patches. This is the stable mainline version released by Linus Torvalds.

not exist *unused/dead code* in the final kernel binary. If a subsystem is disabled it will not be part of the resulting binary.

On page 14 Ron Pierce writes about temporal partitioning according to [Rus99] and concludes: "Temporal partitioning is therefore rather weak in Linux". This is not true because there exist interesting projects which deal with this issue. Of course, temporal partitioning is very important for real-time applications and systems, therefore most of the interesting ideas come from this part of the Linux community. For example RT-Linux [45] uses a small real-time kernel that coexists with the standard Linux kernel. According to Yodaiken [1], in RT-Linux a simple real-time executive runs a non-real-time kernel as its lowest priority task. On the other hand there exists RTAI [44] [BBD+00], which is another Linux real-time approach that guarantees temporal partitioning of applications and it is successfully used in real-time critical systems like TTA [KB03] (Time Triggered Architecture), which uses TTP [Gro04, KG93] (Time Triggered Protocol).

On page 16 Ron Pierce writes that most if not all SIL4 applications have hard real-time behaviour defined and that "the conclusion reached is that Linux is not suitable for HRT (hard real-time)". This conclusion is questionable because Linux has support for many features that are enforced by real-time critical systems. For example Linux supports a second timer system besides the traditional `kernel/timers.c`. This high resolution timer (hrtimer[2]) subsystem brought various advantages for applications that use nanosleep, POSIX-timers and *itimer* interfaces. The hrtimer subsystem was verified according to POSIX-timer tests in practice. As mentioned before, RT-Linux can be used (and it is) for real-time critical applications. According to Victor Yodaiken [1] from the New Mexico Institute of Technology, the interrupt latency on a 486/33MHz PC was measured well under 30 microseconds, which is close to the hardware limit. On the other hand the main kernel line introduced real-time features over the past years. For example the official kernel 2.6.18 introduced priority inheritance support to prevent priority inversions, and extensions to the generic interrupt handling layer across all architectures which enable predictable and consistent performance in Linux [3].

On page 17 of Pierce's report he writes: "There appears to be no defence against a peripheral system or device which is generating an excessive interrupt rate, which is the main way in which an operating system can be overloaded". This can be avoided in the way RT-Linux handles interrupts because interrupts are only forwarded from the RT-Linux kernel to the normal kernel if there are no real-time tasks.

In the chapter "Assessment of Linux provision" Ron Pierce writes: "In particular there is no direct provision of accurate watchdog timers which will take action if a process has not responded within a given deadline". Linux provides support for hardware and software watchdogs. This support can be enabled via `Device Drivers -> Character Devices -> Watchdog Cards`. Various hardware watchdog cards are supported. On the user side the behaviour for shutdowns or reboots can be controlled via a watchdog deamon [46]. A `/dev/watchdog` is created and it is possible to monitor processes, therefore Ron Pierce's argument does not hold. For example if someone wants to monitor the Apache (web-server) process, it can be done by the directive `pidfile = /var/run/apache2.pid`. The watchdog program has other interesting features, too. For example it is possible to monitor network interfaces, memory-usage or maximum-load scenarios. Even the fitness of a network host can be monitored. If a problem is examined there are different methods of handling them. There are the traditional ones like shutdown or reboot, but a *repair binary* can be specified, too. Most of them are shell-scripts which can be used to restart a failed service.

[2] /usr/src/linux/Documentation/hrtimers/hrtimers.txt

On page 27 Ron Pierce writes that "Linux systems will require the attention of a system administrator", and as an example for it he uses the removal and archiving of old log files. As this might be true in general, there are differences in detail. GNU/Linux is used for several embedded projects where administration is not required. For example digital video recording set-top device like TiVO [51], Relook [42], and Dreambox [8] [Gäb06] use GNU/Linux and it does not require administrative interaction. A second example is the use of GNU/Linux in network routing devices from Linksys [14]. If the router is configured according to the users needs, no further interaction or administration is necessary. Of course, these systems are not used in a safety-critical context, but these examples show that GNU/Linux based systems do not necessarily need this administrative attention claimed by Pierce. In addition there exist tools which automate the process of archiving and rotating log files. There exists a tool named *logrotate*, which is able to automate this process. According to its configuration the program archives the old log file at a point in time, creates a new log file and if it is required, the program sends a signal to the program which owns the log file. Besides from that, old log files are removed.

In the chapter "Testing" on page 40 Ron Pierce writes: "Another problem is that the testing carried out to date by the LTP and others generally is black box or requirements based rather than white box or structural testing. The extent of code coverage of the tests is therefore unknown". As presented in section 2.5 and according to the LTP project page [30], LTP does provide tools which can be used for code coverage analysis, therefore the situation changed to the better. In the same chapter Ron Pierce writes that operating system testing is different form application testing in several ways. In general he is absolutely right, but there have been enormous efforts during the last years which make it much more comfortable (or at least possible) to test Linux. For example there is a project called *User Mode Linux* (UML) which makes it possible to run Linux (the kernel) as a normal user-mode process inside a GNU/Linux system [Dik06]. This makes debugging for developers much easier. If the kernel fails (often referred as "the kernel panics"), it is only a normal process that fails but there is no need to endanger the whole system, which uses a stable kernel as its base. There is a second project, the FAU-Machine [HWS04], which provides simple methods for injecting faults into the kernel that runs in a virtual machine.

4.2 COTS Components in Safety-Critical Systems

The paper "COTS components in safety-critical systems" [Kri02] was written by Linda Kristiansen in 2002 and is about the use of COTS components in safety-critical systems and what makes it difficult to use them.

There is an interesting assumption in the abstract of the paper. Kristiansen writes, "COTS components are hence black boxes to their users, and the greatest concern about the component is not what you know it can do. The things you do not know that the component is capable of are far more difficult to handle". While this is an important argument that holds most of the time, because as it is known, in most cases the 'C' in COTS implies that the source code of the component is closed and not known because companies have commercial interests and do not provide source code. On the other hand Open Source software has the advantage that the source is known and therefore someone with the ability to read and understand this source code knows perfectly well what this component is capable. In this case the development model used by the GNU/Linux community has an advantage over the traditional closed source ones. This conclusion is also made by Kristiansen because on the next page she writes: "The problem of

lack of insight into the programming code present in most COTS components could be avoided by using Open Source Software".

On page eight Kristiansen writes that "the other prospective of unfamiliarity is that the software developers may have insufficient knowledge about the problems the system is supposed to control. They lack the knowledge of the specific industry area". Linux kernel development is done by well known programmers who have a long experience in the field of safety-critical applications. Many of those programmers work for companies that are known for the development of safety-related software.

On page 12 Kristiansen writes that COTS components have a wide range of users who provide a large test base. This argument holds because it is a simple equation that the more users, in this case testers, a component has, increases the possibility of finding errors. Especially in the market of embedded systems, Linux has a broad user and developer base. The main arguments for Linux for embedded systems was its portability and the reduced costs of a COTS component. Most of the embedded devices that run a Linux kernel or parts of the GNU/Linux system do not have safety-critical requirements like internet routing devices or Linux powered smart phones, but the area of safety-critical applications and devices can benefit from the huge amount of Linux users and developers in the sector of non critical embedded systems. That is the same conclusion that Kristiansen makes in her paper when she writes "Since many have used the product, the likelihood of undetected errors is small". According to her and to Boehm [BA99], frequent product updates can have drawbacks, because commercial vendors typically provide only support for the newest release which forces users to upgrade to a component that is not that well tested. This is an interesting statement and it perfectly makes sense in the field of commercial products, because commercial vendors or vendors of proprietary software tend to encourage their users to update and upgrade, because that is the way they make money. This argument holds in the field of some well known GNU/Linux distributions (see section 3.5.2), because the distributors want to make money. In defence, it should be mentioned that most of these distributors offer special long term support where they provide critical updates but do not force someone to use a complete new, and therefore untested version. It is very difficult to distinguish where a software patch only corrects a software bug without interfering with safety assumptions and where the component has to be re-validated. In general, the field of Free and Open Source (FOSS) COTS has a big advantage over the traditional commercial ones. There is nobody that would force a developer to upgrade and as long as there is interest in a component, it will be developed. This can be explained on the example of the Linux kernel series 2.4. The mainstream development nowadays is done in the series 2.6, but the old one is still important and interesting for developers of embedded systems because, the hardware requirements for the old series is much lower than for full featured 2.6 kernels. As it is known, kernel development is done by the paradigm of Open Source and therefore nobody tries to force someone to the newer 2.6 series. If the kernel would be proprietary software the older kernel tree would not exist nowadays. So there is a market for the older tree, because of its lower hardware requirements and therefore it is still not forgotten. Of course, the update cycles are very long, but still new versions of the 2.4 kernel are done. And the long release cycle does not implicitly mean that that it is dead code, it is a sign that the code is stable, too.

The advantage of the Open Source paradigm is shown on page 15 where Kristiansen writes "COTS components often appear as black boxes the developers have to deal with. This black-box association is what causes the problem". Further she writes "thus, the developer of a safety-critical system can analyse his work, but where COTS components are involved the reasoning becomes weaker, since he cannot argue for the internal functions of the COTS component". COTS components that follow an Open Source development model have two main advantages

to the former quoted arguments. First of all the developer is able to review, test, and maybe certify the code of the COTS component because he or she does not need to treat the component as a black-box because the code is available. The second advantage is that this code review is already done by other developers which use the same component. Of course, someone has to be careful with this assumption, but the probability for it is much higher than for a traditional COTS component. This is kind of an extension to the former mentioned argument that the more people use a COTS component the less errors will be undetected. The more people have the *possibility* to review code, the more it will be reviewed and therefore errors should be found even faster.

On page 16 Kristiansen is very sceptic that COTS components should perform critical operations. She shows a figure similar to figure 4.1 to demonstrate her conclusion.

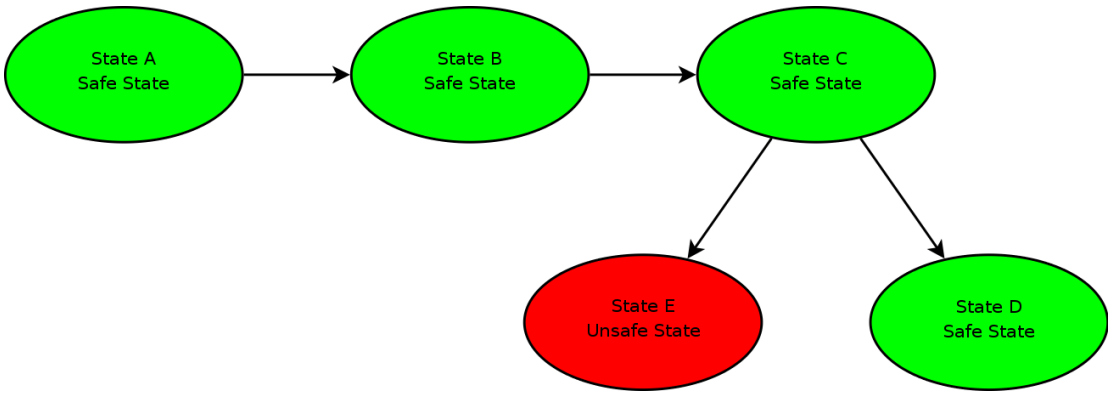

Figure 4.1: Event chain for safety-critical systems with COTS components

The figure shows states from A to E where the states A and B are non critical because they always have a safe state as its predecessor. According to Kristansen state C should not be controlled by a COTS component because it can lead to the unsafe system state E. The argument for this decision is the following: "The reason for advising against the use of COTS in this situation is that the COTS component is more difficult to control than custom written code would be in this situation". While this makes sense from Kristiansen's point of view, it does not hold if the COTS component would have been a FOSS component because FOSS COTS components are not categorical more difficult to control than their self written counterparts.

On page 27 Kristiansen writes that white-box testing is not effective and often not possible with most COTS components. As described in section 3.3.2, white-box testing has several advantages that give much more confidence than other testing approaches like black-box testing. Kristiansen is right that white-box testing is difficult and for proprietary components it is impossible to do white-box testing, but in the context of Open Source COTS components, it is very well possible. Therefore the Open Source paradigm can solve the problem and gives the designer the power to do proper and sufficient testing of the system or component.

When Kristiansen writes about fault injection she is right when she is sceptic that it is applicable for traditional COTS components. One solution is to alter the output of a component on its output interface and use this output (with the injected faults) as input for a second component. This simple method of fault injection can be done for Open Source COTS components too, but there exist much more powerful alternatives to this approach. A simple but powerful method

is provided by the *gdb* [12] (The GNU Project Debugger). This tool can execute programs and run them step by step. The *gdb* offers traditional debugger features like setting breakpoints and changing and displaying the values of variables. The *gdb* is able to execute debugging scripts, which makes it a perfect tool for fault injection. Kernel level fault injection and debugging is much more complicated than traditional software debugging, but there exist several tools which support Linux kernel developers. First of all there exists a special tool that is similar to the gdb which is named *kgdb* [29] (Linux Kernel Source Level Debugger). On the other hand several forms of virtualisation can help with kernel level debugging and fault injection. There exists a special project which is designed for Linux fault injection which is the *FAU Machine* [HWS04]. One of the most used projects for fault injection is *User Mode Linux* [55] which lets someone execute the kernel as a normal process in the user mode. A project that was recently integrated in the mainstream kernel is *lguest* [25] [Rus07], which provides similar features as *User Mode Linux* but is much more light weight.

On page 52 Kristiansen has several doubts on FOSS components, because she writes: "Another possible problem with OSS is the rapid change in the components. The OSS components are under constant change, because there are many developers working to improve them in one way or another. Some of the official versions of the components contain errors or bugs, and these errors will not be corrected until users describe them, and someone contributes a version where the problem has been fixed". In general bugs are only fixed if someone provides a fix for them, that is the nature of bug fixing and not a FOSS problem. Bug fixing in the Open Source community is even superior than in the traditional software developing models. For example, most know examples of the Microsoft Windows operating system or the Microsoft Internet Explorer where bugs have not been fixed for a very long period. Besides from the enormous number of security related bugs that exist for these two products, the Open Source community is faster in fixing bugs. Concerning major Open Source products a fix for a bug exists hours after the reporting of the bug itself. For example take a look on famous Open Source projects that are stable and secure even if there are hundreds of developers. Consider the Apache web-server, the Linux kernel, the Samba server, or the Firefox web-browser. All of them are developed by a huge number of programmers and they are known as stable products. Of course it is hard to compare security and safety, but developing secure products is a difficult task and it shows that the Open Source development model can provide secure software. In fact it is not a problem of the development model, it is a problem of quality management and this quality management works very well in the Open Source software world.

On the same page Kristiansen writes: "Therefore, you wish to use OSS components in safety-critical systems, it would be wise not to use a young version of a product (typically in the range of versions 1.1 to 2.9), and instead aim for the more mature versions". That is a good and universally valid advice, but what Kristiansen writes about version numbers is not true. There exist lots of tools that have version numbers below 2.9 and are rock stable and mature. Some of them are developed for decades and have not reached a version number above 2.9. For example the Linux kernel is at version 2.6, or the old stable 1.x tree of the Apache web-server series is still used on lots of servers nowadays. Most of the GNU tools have very low version numbers, like the GNU *grep* command which is at version 2.5 or the GNU *man* system which is at version 1.6. There are many ways how the individual projects give there version numbers, but there is no general advice. And there is a reason why these products have relative low version numbers. It is because the developers do not need to raise the version number for marketing reasons. For FOSS software developers it does not make sense to call their newest product "XP" or raise the first digit of the version by every minor bug fix.

4.3 Further Important Work

The previous sections discussed individual papers whereas the following sections concentrate on major important topics in the context of Linux in safety-critical applications.

4.3.1 Linux and Real-Time

As mention in section 4.1 the real-time capabilities of Linux sometimes get criticized. At the time when Ron Pierce wrote his paper (Linux 2.4.x), there were in deed some shortcomings in this part of the kernel. But the kernel series 2.6 brought some inventions that make the kernel much more real-time fit. There exist so called *rt-preempt* [RH07] patches that help the kernel in several ways. First, the patches allow that the kernel is preempted in nearly all sections with a few exceptions. It is obvious, that the possibility to preempt the kernel helps the overall timing of the kernel. The second important invention was the inclusion of high resolution timers (*hrtimers*). The combination of these two things give the Linux kernel hard real-time behaviour [36].

To support this claim some companies recently made all kinds of latency tests. One test was done by Alexander Bauer [Bau07] where he examined the real-time capabilities on a PowerPC and on an ARM board. His results are summarized in table 4.1 and table 4.2.

Table 4.1: Cyclictest on MPC5200 (PowerPC) and PXA270 (ARM)

Cyclic Test	PPC (266 MHz)	PPC (400 MHz)	ARM (260 MHz)	ARM (520 MHz)
Idle	60 μs	45 μs	550 μs	450 μs
Under stress	120 μs	95 μs	600 μs	550 μs

Table 4.2: Interrupt latency (IRL) on MPC5200 (PowerPC) and PXA270 (ARM)

IRL	PPC (266 MHz)	PPC (400 MHz)	ARM (260 MHz)	ARM (520 MHz)
Idle	9 μs	8 μs	85 μs	70 μs
Under stress	11 μs	10 μs	112 (max. 260) μs	90 (max. 150) μs

A team of *ABB AS Robotics* and *ABB Corporate Research* made Linux real-time capability tests [MSR07], too. The team used the following hardware:

- Custom made for ABB and based on the Lite5200 evaluation board from Freescale.
- Freescale processor MMPC5200B, which is an embedded PowerPC processor based on the MPC603e Core.
- 750 MIPS running at 400 MHz
- 64 MB DDR SDRAM and 32 MB Flash

Table 4.3: Write Latency Test [MSR07]

Latency	Min	Max	Avg	Jitter
Idle	457.136 μs	539.530 μs	499.508 μs	82.394 μs
Under stress	439.285 μs	537.075 μs	494.517 μs	97.790 μs

Table 4.4: Read, Write, and Interrupt Latency Test [MSR07]

Latency	Min	Max	Avg	Jitter
Idle	263.800 μs	434.458 μs	296.240 μs	79.658 μs
Under stress	266.043 μs	364.067 μs	286.358 μs	98.024 μs

The results of the benchmarks that were run on Linux-2.6.21-rt3 kernel can be found in table 4.3, and table 4.4. The situation under load was created with a tool called stress [47], and the defined sleep time was 500 μs.

From these results ABB comes to the following conclusion: "During the course of this evaluation we have validated and benchmarked certain real-time performance parameters in the context of paint robots. The results are optimistic and the prospects are bright for the future with Linux" [MSR07].

There is a second important point that shows the advantage of Linux over other, more traditional used, real-time operating systems: "The ability to be able to run a real-time application on the same processor as other standard applications is a winning combination. This is really what favours Linux as a real-time operating system compared to other dedicated real-time operating systems" [MSR07].

4.3.2 Linux, Safety, and COTS

As it has been worked out in this thesis there is a lot of progress in Linux kernel development and some main points of criticism have been removed or at least progress has been made. The paper "Linux for Safety Critical Systems in IEC 61508 Context" [Gui07] written by the Linux kernel developer Nicholas Mc Guire shares most of the arguments that have been presented in this thesis and he summarizes it in the following way:

- Advances in the kernel software life-cycle
  - Introduction of subsystem maintainers
  - Developer branches and arch branches for early testing of features (i.e. arm.linux.org.uk)
  - Well defined experimental tree (-mm) and the introduction of the merge-window
  - Early testing in the release candidates (-rcX)
  - Long term road-maps for feature introduction (i.e. RT-preempt is being merged in steps since early 2.6.x)

- High-level management elements introduced in 2.6 - beyond LKML
  - Annual kernel summit for strategic decisions
  - Domain specific groups (i.e. CELinuxForum Architecture Group for consumer electronics)
  - Auditing introduced for critical API (i.e. raw_spin_lock usage)
  - Change-log management
  - Improved maintenance of kernel specific information (i.e. lwn.net, kerneltrap.org, changelogs)
- Testing and validaton
  - Critical resources include built-in tests (i.e. RCU torture test, lock-dependency validator), especially in 2.6.x the development of built-in tests have resulted in detection of a large number of bugs without that these ever struck in the field.
  - Linux Test Project (LTP) providing a high-level test-coverage of the Linux kernel providing roughly 30000 test of the Linux OS (ltp-20070831).
  - Crackerjack - kernel code coverage test-suite (crackerjack.sf.net)
  - `http://test.kernel.org` - Autotest is a framework for fully automated testing of the latest Linux kernel release - published online and available to the public.
  - POSIX test-suit

A relatively new organization, where Nicholas Mc Guire is involved, is the *Open Source Automation Development Lab* (OSADL). Their goal is to promote the usage of Open Source software in the context of machine and plant control systems and to help to meet the demand of the automation industry [33]. There are several ambitious projects on the roadmap which cover the most important aspects of this thesis. The focus in on two major aspects of Linux. The first one is the *real-time* aspect and the second one is the *certification* of Linux. Driven by kernel programmers, members of the industry, and by academic members, this project has the likelihood to push Linux in the longed for position. The following OSADL projects [33] focus on the contents of this theses:

- Implement real-time capabilities in the mainline Linux kernel
- Prepare mechanism to allow for the certification of the Linux kernel
- Provide a compatibility layer to use RTDM-based (Real-Time Driver Model) drivers under RT-Preempt
- Support the development of the KVM (kernel-based virtual machine) in such a way that running a guest system does not interfere with the real-time capabilities of the host system.
- Provide support for real-time Ethernet boards to the Linux kernel
- Add real-time Linux support for Coldfire (68knommu) to RT-Preempt

5 Improving Linux for Safety-Critical Applications

The previous chapters have shown an appraisal of the current status of Linux and the GNU/Linux operating system. The following sections give two examples how to improve features of Linux to make the kernel more suitable for safety-critical applications. It is important to note that these examples are showcases that demonstrate the possibility of improving, testing, and extending the Linux kernel or features of it. Section 5.1 provides a fully automatic test-suite for in-kernel RAID-1 systems, also known as "Software-RAID". Section 5.2 shows the implementation of a wrapper file system that implements a layer of safety over already existing file systems.

5.1 A Test-Suite for Kernel Level RAID-1 Systems - raid1test

Raid1test is a test suite of programs and scripts that test a RAID (Redundant Array of Inexpensive Disks) level one environment. For testing the RAID system two computers running the GNU/Linux operating system are needed. RAID systems by them self do not provide any safety relevant features so it is important to explain their context for safety critical systems. RAID (level one) improves the availability of a system by writing data that is written to a special device to two, or more, physical hard drives. If one disk stops operation the system is able to read the data from the second one and can continue to operate. Therefore, as said before, RAID improves the *availability* but not the *safety* in the traditional sense. But availability can be part of the safety argumentation and availability is part of safety in real world systems. For example, a plane that is in the air cannot switch to a state where it stops all of its operation, which can be possible in other systems. If the plane would stop the operation it would crash. In such a system availability is part of the safety argumentation. For example, if the system is designed in a redundant way, one part can adopt the operation of a failed part. RAID-1 provides this redundancy on hard disk level and should be considered in the context of safety.

As stated before, RAID improves the availability and in many cases the safety, but RAID level one has one important problem that is crucial in a safety relevant context. RAID-1 mirrors data to multiple disks and can mask a disk failure by reading from functional disks, but it does not have an ability to check if two data sets are equal during read. Therefore, if one disk does not fail completely, but sends malicious data, the whole system is in trouble, which is shown in figure 5.1. One disk sends a bad block of data, which is shown in red, but the whole RAID system is affected

by one disk. RAID 1 reads data from different disks to improve the read speed of the RAID system.

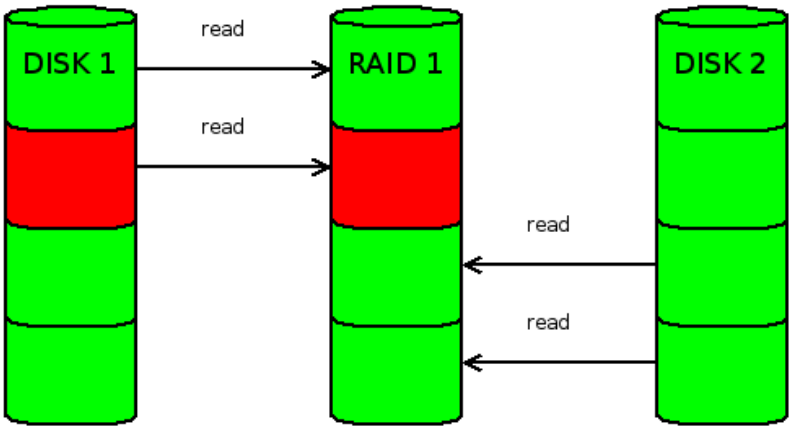

Figure 5.1: Read from RAID-1 with one malicious hard disk

The conclusion is that RAID systems have the ability to improve safety, but there is something missing, a check if the data is consistent during read. Section 5.2 will provide a possible solution for this matter.

The first step in helping RAID-1 to be relevant for safety-critical systems is to test the current status of RAID-1. Only if someone can be sure that the underlying, existing RAID infrastructure in the Linux kernel is working according to the specification, additional steps can be taken to improve the safety of RAID-1.

This section concentrates on RAID-1 testing, whereas section 5.2 shows safety relevant improvements for file systems.

5.1.1 Implementation

To make *raid1test* a full featured, automated test suite a combination of hard- and software was needed. Section 5.1.1.1 describes the hardware parts of the test suite and section 5.1.1.2 concentrates on the software side of the test suite. The first task was to find a set of test cases which test all the relevant features of a RAID-1 system. The *Linux Test Project*[32] specifies a test plan for RAID-1 systems which was used as a guideline for this project.

5.1.1.1 Hardware

The first thing noted from the test plan was that there has to be a solution to switch on and off hard disks to simulate failed disks. To run the tests in an automated way, this has to be done by the test suite itself. It is not an option to manually switch on or off the power of a hard disk. The solution was to split the test suite into a server and a client part. The server, called *master*, is the head of the system that delegates orders to the client (*slave* node) and keeps log of the

current test and the state of the test suite. This server can physically switch off hard disks of the client via a special hardware device that is controlled by a kernel module. The kernel module controls output pins of the computer's parallel port which turns on or off relays. These relays are connected to the current line of a hard disk. A schematic of the test suite is shown in figure 5.2.

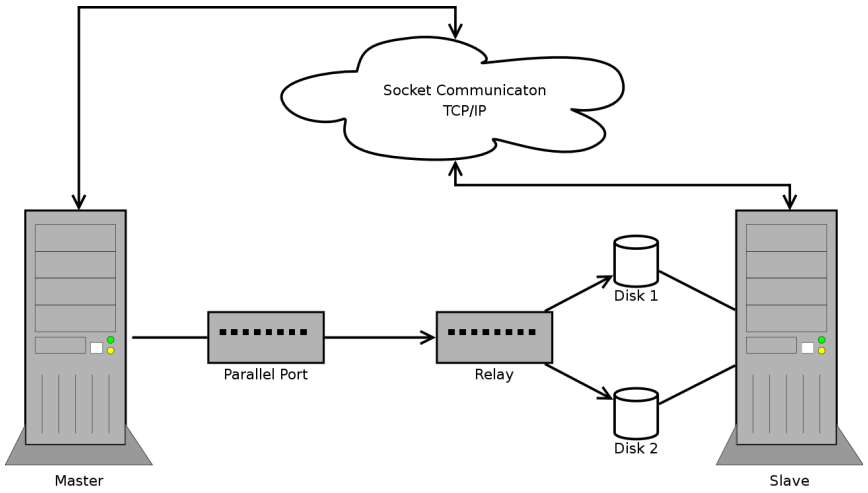

Figure 5.2: Schematic of the test suite

The relays switch the power lines of the hard drives and simulate a breakdown of a hard disk. The schematic of the circuit is shown in figure 5.3.

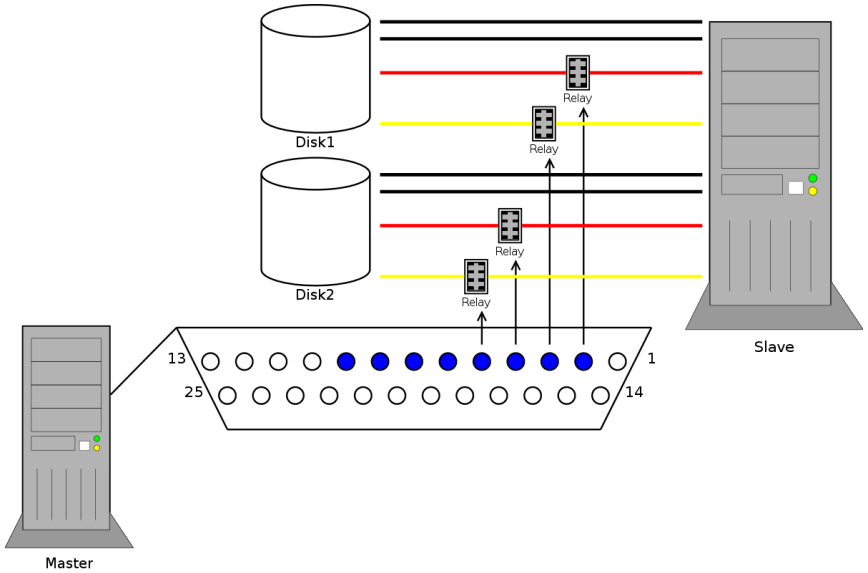

Figure 5.3: Schematic of device that turns off and on hard disks

5.1.1.2 Software

The test suite has the following features, that will be discussed later in this section.

- Test suite has a client/server architecture.
- TCP/IP socket communication between client and server.
- Client can be forked to background.
- Full automated testing without manual interaction. This is provided by special hardware (section 5.1.1.1) and by distribution specific start-up scripts.
- Tests use strong checksums (SHA1) to verify the integrity of files.

The server keeps track of all events, which has two major advantages. The first one is that the client can reboot itself, which is necessary for several tests, without being affected to know in which state the client was before the reboot. After an reboot the client initiates a connection to the server and the server tells the client which test, or which part of a test, has to be done. There is a second advantage of this architecture, which is even more important. It is always better to trust a second, independent node (the server), than trusting the node that is under test.

Client and server communicate via standard UNIX sockets, which has the advantage that the ordinary Ethernet hardware can be used. The communication is separated from the client and server implementation in a communication API. This API is used by the server and the client to communicate to each other and to log the communication. Listing B.1 shows the header file (`communication.h`) of the communication API. The source is documented with doxygen [7] and gives an overview of the communication API.

As an example for the implementation of a communication function, the next listing (listing 5.1) shows the source code of `make_socket` which is used to create a new socket.

Listing 5.1: Source code for making a new socket

```
1  /**
2   * @brief Create a new socket
3   *
4   * The function creates a new INET socket on a specified port.
5   *
6   * @param port The socked is bound to this port address.
7   *
8   * @return Socket file descriptor in case everything is okay.
9   */
10 int make_socket(uint16_t port)
11 {
12     int sock;
13     struct sockaddr_in name;
14
15     /* Create the socket. */
16     sock = socket(PF_INET, SOCK_STREAM, 0);
17
```

```
18      if(sock == -1)
19      {
20          perror(__func__);
21          exit(EXIT_FAILURE);
22      }
23
24      /* Give the socket a name. */
25      name.sin_family = AF_INET;
26      name.sin_port = htons(port);
27      name.sin_addr.s_addr = htonl(INADDR_ANY);
28
29      if(bind(sock, (struct sockaddr *) &name, sizeof(name)) == -1)
30      {
31          perror(__func__);
32          exit(EXIT_FAILURE);
33      }
34
35      return sock;
36  }
```

The function is called with one argument which is the *port* of the new socket. The `socket` call returns a new socket file description. In this example it creates an IP version 4 socket (`PF_INET`) socket that provides sequenced, reliable, two-way, connection-based byte streams (`SOCK_STREAM`, `man 2 socket`). After that the structure *name* gets initialized with the protocol family (`AF_INET`), the port of the socket and the internet address (`sin_addr`). The `hton` functions converts from "host byte order" to "network byte order" (`man htons`). The next step is to call `bind` which is the clue between the socket (*sock*) and the *name*. The `bind` function assigns a name to a socket (`man 2 bind`). Before that, the socket exists, but has no name assigned. If everything is okay, the function returns the file descriptor of the new socket, which is an ordinary integer variable.

The next implementation detail is that the client can be forked to the background which is a nice way to let the client run as daemon. This can be used to start the client via GNU/Linux start-up scripts, that are provided in the source code package. Running as daemon guarantees that the whole test suite can be run fully automatic. For example, the server is started up and initialized and then the client gets booted. Turing boot up the start-up script for the client is executed and the client gets forked to the background. It connects to to the server (specified via internet name and port in a configuration file or via command line arguments) and begins the communication and testing. After an reboot, the client connects again and begins to run the remaining tests.

Listing 5.2: Source code to run the client as daemon

```
1  /** @brief Run programm as daemon.
2   *
3   * This function allows to run the program in background (as daemon).
4   * It closes unneeded file descriptors and forks the process
5   * to the background.
6   * It creates a lockfile and writes the PID to it.
7   */
8  void daemonize(void)
9  {
```

```
10      int i, lock_fp;
11      pid_t pid;
12      int pidlength = 15; /* thats enough */
13      char *s_pid = NULL;
14      if(getppid() == 1) return; /* already a daemon */
15
16      pid = fork();
17      if (pid == -1) /* fork error */
18      {
19          perror(__func__);
20          exit(EXIT_FAILURE);
21      }
22
23      if (pid > 0) /* parent exits */
24      {
25          exit(EXIT_SUCCESS);
26      }
27
28      /* child (daemon) continues */
29      setsid(); /* obtain a new process group */
30
31      /* close descriptors */
32      /* for (i = (sysconf(_SC_OPEN_MAX) - 1); i >= 0; i--) */
33      for (i = (sysconf(_SC_OPEN_MAX) - 1); i > 1; i--)
34      {
35          close(i);
36      }
37      /* keep stdin/stdout open, we need them to read output from
38       * mdadm */
39
40      umask(027); /* set newly created file permissions */
41      chdir(RUNNING_DIR); /* change running directory */
42
43      lock_fp = open(LOCK_FILE, O_RDWR | O_CREAT, 0640);
44      if (lock_fp == -1)  /* cannot open */
45      {
46          perror(__func__);
47          exit(EXIT_FAILURE);
48      }
49
50      if(flock(lock_fp, LOCK_EX) == -1) /* cannot lock */
51      {
52          perror(__func__);
53          exit(EXIT_FAILURE);
54      }
55
56      s_pid = (char *) malloc((sizeof(char) * pidlength) + 1);
57      if(s_pid == NULL)
```

```
58      {
59          perror(__func__);
60      }
61      else
62      {
63          snprintf(s_pid, pidlength + 1, "%d\n", getpid());
64          /* record pid to lockfile */
65          write(lock_fp, s_pid, strlen(s_pid));
66          free(s_pid);
67      }
68
69      signal(SIGCHLD, SIG_IGN); /* ignore child */
70      signal(SIGTSTP, SIG_IGN); /* ignore tty signals */
71      signal(SIGTTOU, SIG_IGN);
72      signal(SIGTTIN, SIG_IGN);
73      signal(SIGHUP, signal_handler); /* catch hangup signal */
74      signal(SIGTERM, signal_handler); /* catch kill signal */
75  }
```

First the program checks if the parent process has a PID (process ID) equal to 1. If this is the case the parent process is the *init* process. This process is the first processes started by the kernel during boot up and has the PID 1. If the parent process has PID 1, the function returns because it is already a daemon. The next step is to fork (`man 2 fork`) the process. Fork creates a new process by duplicating the calling process. After the call, there exist two equal processes. The return value of `fork()` determines in which process the following code will be executed. If the return value is -1, an error occurred and the forking went wrong. If the return value is greater than 0, it is the parent's thread of execution. In this case `exit()` is called to end the program and therefore the original process ends. The new process, the child, now takes several actions which are necessary to run as daemon. First of all it calls `setsid()` to obtain a new process group. This is necessary because processes inherit their group from the parent. A real daemon should be unaffected by other processes and therefore it is a good practice to call `setsid()`.

Daemon processes usually run with root-privileges and therefore someone should take attention with which rights files get created. In this case the `umask` is set to "027" (`man umask`).

It is a good practice to run the server in a specific directory independent of the directory it was started. This is done via the `chdir()` call.

To ensure that a daemon runs only one time, *file locking* is used. The first daemon locks a specific file and when it terminates the lock will be released. This is done via a call to `lockf()`. As a bonus the PID of the running process is recorded to this locked file. This makes determining the PID easier for the administrator.

The rest of the function handles so called *signals*. Signals are used to communicate to a process. For example if a user types "CTRL-C" the signal *SIGTERM* is sent to the process. If the process has a signal handler it decides how to handle the signal. In his example the process ignores signals from the child processes and terminal signals. Hangup and termination signals get handled by a signal handler. For a list of signals and their meaning refer to `man 7 signals`.

The following list gives an overview about additional functions that the slave node provides. It does not make sense to discuss every function in detail, therefore only a short overview is given.

The source code is available an documented, therefore it is up to the interested reader to get further implementation details.

- print_help: Prints information about command line options and arguments.
- signal_handler: Handles signals passed to the programm (`man 7 signals`)
- daemonize: Run the program in background, as a daemon.
- reboot_system: Takes care about clean up and reboots the system.
- read_back_data_and_check: Reads back data from the RAID and checks it.
- generate_new_data_and_check: Generates new data and checks it.

The most important functionality is the creation and verification of data, therefore the function `generate_new_data_and_check` is shown in listing 5.3 and discussed afterwards.

Listing 5.3: Source code to generate and test files

```
1  /** @brief Generate new data and check it.
2   *
3   * This function generates new data and reads back this data from
4   * the RAID1 system and transfers
5   * the checksum to the master node.
6   *
7   * @param sockfd Socket file descriptor to write to.
8   * @return 0 in case the checksums are ok
9   * and could be transfered, else -1.
10  */
11 int generate_new_data_and_check(int sockfd)
12 {
13     int retval;
14     char *checksum_read = NULL, *checksum_write = NULL;
15
16     /* generate new data */
17     log_message(writeto, "generating files...");
18     retval = write_files(testdir, file_count, byte_count,
19                          &checksum_write);
20     if(retval != file_count)
21     {
22         snprintf(buffer, BUFFER_SIZE,
23                  "%s: Could not write all files (%d/%d)\n",
24                  cmnd, retval, file_count);
25         log_message(writeto, buffer);
26         return -1;
27     }
28
29     sprintf(buffer, "checksum_write: %s", checksum_write);
30     log_message(writeto, buffer);
31
```

```
32      log_message(writeto, "reading files...");
33      retval = check_files(testdir, file_count, &checksum_read, 0);
34      if(retval != file_count)
35      {
36          snprintf(buffer, BUFFER_SIZE,
37                  "%s: Could not read all files (%d/%d).\n",
38                  cmnd, retval, file_count);
39          log_message(writeto, buffer);
40          free(checksum_write);
41          return -1;
42      }
43
44      sprintf(buffer, "checksum_read: %s", checksum_read);
45      log_message(writeto,buffer);
46
47      if(strcmp(checksum_write, checksum_read) != 0)
48      {
49          snprintf(buffer, BUFFER_SIZE,
50                  "%s: checksums differ!\nwrite: %s\nread: %s\n",
51                  cmnd, checksum_write, checksum_read);
52          log_message(writeto, buffer);
53          free(checksum_write);
54          free(checksum_read);
55          return -1;
56      }
57
58      send_message(SLAVE_OK, sockfd);
59
60      /* wait for get checksum */
61      read_and_log(buffer, sockfd, writeto);
62      if(strcmp(buffer, M_GET_CHKSUM) != 0)
63      {
64          sprintf(buffer, "Server sent wrong response in line %d\n",
65                  __LINE__);
66          log_message(writeto, buffer);
67          free(checksum_write);
68          free(checksum_read);
69          return -1;
70      }
71
72      /* send checksum */
73      send_message(checksum_read, sockfd);
74
75      /* wait for got checksum */
76      read_and_log(buffer, sockfd, writeto);
77
78      /* now we can free checksums */
79      free(checksum_read);
```

```
80      free(checksum_write);
81
82      if(strcmp(buffer, MLGOT_CHKSUM) != 0)
83      {
84          sprintf(buffer, "Server sent wrong response in line %d\n",
85                  __LINE__);
86          log_message(writeto, buffer);
87          return -1;
88      }
89
90      /* transfer successful checksums are freed */
91      return 0;
92  }
```

The function tries to generate new data (`write_files`) and stores the checksum of these files. If the generation was successful and the data could be read from the RAID and the checksum is still the same, the client sends the checksum in a readable string to the master node which stores it. After that the RAID gets manipulated, for example a disk is turned off and a call to `read_back_data_and_check` is done. This function reads back the files generated and sends the checksum to the master. The master compares the two checksums and if they are equal, the RAID worked as expected.

5.1.2 Configuration

Information about the basic installation of a RAID-1 system and the installation and configuration of the *raid1test* test suite is given in this section. Both, the master and the slave node, need an operating system that has the ability to compile software. All the further steps have been tested on a *Ubuntu GNU/Linux 8.04 LTS* system with the latest patches and fixes available at writing this document.

5.1.2.1 Hardware

First of all, someone needs two PCs, which will be called *master*, and *slave* from now on. The master is the control node and does not need to have non standard hardware. As the slave, it needs to have a network interface card (NIC) to communicate to the other node. The configuration of the network will not be covered in this document, it is an assumption. If the master should be able to switch off an on the hard disks in the slave node, then the master has to have a parallel port and the hardware device described in section 5.1.1.1. The hardware configuration of the slave node is a bit more sophisticated. It should have one hard disk with the operation system on its own hardware channel (i.e. master on the first IDE channel) and two disks which form the RAID array. It is important that the two disks in the RAID array have *separate* hardware channels. If the two disk are on the same bus, the configuration will not work. For example if the two disks are on the second IDE channel as master and slave (jumper settings), one failed disk can bring down the whole bus and therefore the second disk will be unreachable, too. The Linux documentation project describes this behaviour in the reconstruction section [52] of the software RAID howto.

> "Well, it [reconstruction] usually is [easy], unless you're unlucky and your RAID has been rendered unusable because more disks than the ones redundant failed. This can actually happen if a number of disks reside on the same bus, and one disk takes the bus with it as it crashes. The other disks, however fine, will be unreachable to the RAID layer, because the bus is down, and they will be marked as faulty. On a RAID-5 where you can spare one disk only, loosing two or more disks can be fatal."

This leads to a difficult situation, because the system-disk (with the operating system) needs one separate channel, and the two raid disks need separate channels, too, which makes three channels, but ordinary mother boards provide only two channels. This problem can be solved with a PCI-card that provides additional channels.

Though the configuration of the software RAID on the slave node involves software tweaks, it will be covered in this section. After installing the hard disks as described, and installing Ubuntu on a seperate disk, it is time to configure the RAID itself. It is assumed, that the device nodes of the RAID disks are known. An easy and fast way to get information about the disks is to read the information `dmesg` provides and to scan the disks with `fdisk`. To get information about the disks the command `sudo fdisk -l /dev/sd[a-z]` can be used. After that the individual disks should get a reasonable partition table. The RAID management tool and the kernel can deal with whole disks, but the following example provides information on creating one large RAID partition on the device `/dev/sdb`.

```
1  $ # delete the old partition table
2  $ sudo dd if=/dev/zero of=/dev/sdb bs=512 count=1
3  $ # create new partition
4  $ sudo fdisk /dev/sdb
5  Command (m for help): n <enter>
6  Command action
7     e   extended
8     p   primary partition (1-4)
9  p <enter>
10 Partition number (1-4): 1 <enter>
11 First cylinder (1-1016, default 1): <enter>
12 Using default value 1
13 Last cylinder or +size or +sizeM or +sizeK (1-1016, default 1016): <enter>
14 Using default value 1016
15
16 Command (m for help): t <enter>
17 Selected partition 1
18 Hex code (type L to list codes): fd <enter>
19 Changed system type of partition 1 to fd (Linux raid autodetect)
20
21 Command (m for help): w <enter>
22 The partition table has been altered!
```

Most of this is self explaining but it is encouraged to read `man fdisk` or use the built in *m* command. The second disk should have equal size and partitioning. An easy way to copy the partitions from `/dev/sdb` to `/dev/sdc` is to use `sudo sfdisk -d /dev/sdb | sfdisk /dev/sdc`.

During the test of the RAID-1 array it was recognized, that switching disks on and off can influence the internal order of the device nodes. To be on the safe side, it is a good idea

to map the hard disks to special unique names. This can be done with *udev*. The following example assumes that disks are matched according to their *model* parameter, but everyone is free to match the disks according to his or her preferences. To get information about a disk the command `sudo udevinfo -a -p $(udevinfo -q path -n /dev/sdb)` can be used. The information gathered can be used for custom udev rules. During the test the following `/etc/udev/rules.d/25-names.rules` file was used:

```
KERNEL=="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="SAMSUNG SV2042H  ", SYMLINK+="firstraiddisk%n"
KERNEL=="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="Maxtor 90431U1   ", SYMLINK+="secondraiddisk%n"
```

This will create device nodes like `/dev/firstraiddisk1` for the first partition of the disk that matches the model string defined by the udev rule.

After this is done, the RAID is ready to be configured. This is done in two steps. First the raid is created and then the new raid device gets a file system. It is assumed that the two disks have proper `/dev` nodes.

```
$ # install mdadm if it is not installed
$ sudo apt-get install mdadm
$ # create the RAID device
$ sudo mdadm --create --verbose /dev/md1 --level=1 \
             --raid-devices=2 /dev/fistraiddisk /dev/secondraiddisk
```

If it is planed to use a partition instead of the whole disk, `/dev/firstraiddisk` should be replaced with `/dev/firstraiddisk1` or similar to use a partition.

The second step is to create a file system on the devices, which is done the same way it would be done for a single disk: `sudo mkfs.ext3 /dev/md1`. The RAID tool assumes that the RAID is "dirty" and therefore it syncs the array. The progress can be checked on the command line with `watch -n1 cat /proc/mdstat`. After that the RAID is ready for use. For example, it can be mounted like an ordinary disk (`mount /dev/md1 /raid`).

The RAID tools monitor the array and try to find every disk and partition that can be used as RAID device. During the test this was restricted to the special disks in use. The following example shows the file `/etc/mdadm/mdadm.conf`.

```
# mdadm.conf
#
# Please refer to mdadm.conf(5) for information about this file.
#

# by default, scan all partitions (/proc/partitions) for MD superblocks.
# alternatively, specify devices to scan, using wildcards if desired.
#DEVICE partitions
DEVICE /dev/*raiddisk*

# auto-create devices with Debian standard permissions
CREATE owner=root group=disk mode=0660 auto=yes

# automatically tag new arrays as belonging to the local system
HOMEHOST <system>

# instruct the monitoring daemon where to send mail alerts
MAILADDR root

```

```
20 # definitions of existing MD arrays
21 ARRAY /dev/md1 level=raid1 num-devices=2 \
22        devices=/dev/firstraiddisk,/dev/secondraiddisk \
23        UUID=4330b429:cd7aa906:e368bf24:bd0fce41
```

If this is done the RAID is set up and the *raid1test* test suite can be installed.

5.1.2.2 Software

In order to build software from source, someone has to have various build-tools, libraries, and a compiler installed. On both sides, the master and the client, the `build-essential` package has to be installed and the source code of the raid1test suite has to be present. The following listing shows the steps that are necessary on both sides:

```
1 $ sudo apt-get install build-essential libgcrypt11-dev
2 $ mkdir ~/src
3 $ cd ~/src
4 $ # now copy the tarball to this directory before proceeding
5 $ ls
6   raid1test-0.0.1.tar.gz
7 $ tar -zxvf raid1test-0.0.1.tar.gz
8 $ cd raid1test-0.0.1
9 $ ./configure
```

The *libgcryt*, which is used to generate and verify SHA1 checksums is needed by the slave, but it is checked in the *configure* script, therefore it has to be present on the master and the slave side. After running the *configure* script someone can type `make`, which will build *raid1testmaster*, *raid1testslave*, and gives information about building the kernel module *raid1test.ko*. These three components can be built individual as the following listing shows:

```
1 $ cd src
2 $ # build only the master:
3 $ make raid1testmaster
4 $ # build only the slave:
5 $ make raid1testslave
6 $ # build only the kernel module:
7 $ cd kernelmodule
8 $ make
```

To build the kernel module the Linux kernel header files have to be installed. Select the package that fits the kernel by typing `sudo apt-get install linux-headers`. The `make` command in the `kernelmodule` directory provides extra information for creating the device nodes in `/dev` and information on loading the kernel module. Read them very carefully.

To install the whole suite type `make install` in the root of the source code directory. Note that only *root* can install software.

```
1 $ cd ~/src/raid1test-0.0.1
2 $ sudo make install
```

By default the *raid1testmaster* is configured to "manual mode", which means that it will print information to switch on and off hard disks and it will wait for confirmation, but the disks have to be switched manually. If the special hardware device and the kernel module should be used it is important to set a switch device (`--switchdev`) that is different from the keyword *<none>*. Please read *man raid1testmaster* for more information.

The raid1testmaster has to be started before the raid1testslave is executed. There is a man-page which describes the various options, or just run `raid1testmaster --help`

After running raid1testmaster it is time to run raid1testslave on the slave PC. There are several possibilities like manually starting the program or let it run via *init scripts*. There are several preconditions before running the binary:

- Network connection.
- raid1master has to be started.
- The RAID device has to be mounted

In order to mount the RAID at system start, a line can be added to `/etc/fstab`, that is similar to the following (replace 'X' with your RAID device, and `/raid` with the desired, existing mount point).

```
1 $ vim /etc/fstab
2 /dev/mdX    /raid    ext3    auto    0    0
```

After the raid is mounted it is a good idea to run `raid1testslave --help` or read the man-page. An example program start may look like this, where "master" is the DNS name of the master node:

```
1 $ raid1testslave --name=master --raiddev=/dev/md1 --testdir=/raid \
2                  --disk0=/dev/firstraiddisk \
3                  --disk1=/dev/secondraiddisk -f23 -m42
```

Note that the *disk* argument does not need to be a whole disk. If your RAID is configured to use single partitions, then the options could look like this: `--disk0=/dev/firstraiddisk1`. The former example tests the disks *firstraiddisk* and *secondraiddisk* in the RAID array *md1* which is mounted to */raid*. The *23* files, each *42*MB in size, will be written to the root of */raid*. The *name* of the master can ether be a resolvable DNS name or an IP address.

After starting *raid1testslave* it connects to the master and runs the test cases. During this, the system will be rebooted. The results will be prompted to *stdout*, or to a specified log file. If the *raid1testslave* is started in the way that it writes its output to a log file, it is a good idea to use `tail` to see what is going on.

To automate the whole testing process there are *init scripts* in the source directory which can be used on Ubuntu and Debian systems. Just copy the example scripts to their right place and alter the default configuration.

```
1 $ cd ~/src/raid1test-0.0.1/startup-scripts
2 $ cd debian_ubuntu
3 $ sudo cp etc/init.d/raid1testslave /etc/init.d
```

```
4 $ sudo cp etc/default/raid1testslave /etc/default
5 $ # change the default values
6 $ sudo vim /etc/default/raid1testslave
```

After that *raid1testslave* can be started by typing `sudo /etc/init.d/raid1testslave start`. If the daemon should be started at boot up, create the needed links by typing `sudo update-rc.d raid1testslave defaults`.

5.1.3 Test Results

Each run of the test suite contained 6 tests that are documented in table 5.1.

Table 5.1: Description of test cases run by the suite

Test case	Description
Test 1	Checks if RAID is up and working.
Test 2	Turn off disk 0. Read back and check data from previous test. Write new data and check them while RAID contains only one disk. Turn disk 0 on and reboot system.
Test 3	After the reboot from the previous test check RAID status. Read back data and check it from previous test. Hotadd disk 0. Synchronize the RAID array and wait until it is finished. Read back and check data from synced RAID.
Test 4	Checks if RAID is up and working again.
Test 5	Turn off disk 1. Read back and check data from previous test. Write new data and check them while RAID contains only one disk. Turn disk 1 on and reboot system.
Test 6	After the reboot from the previous test check RAID status. Read back data and check it from previous test. Hotadd disk 1. Synchronize the RAID array and wait until it is finished. Read back and check data from synced RAID.

Table 5.2 shows the test results of the raid1test test suite.

Table 5.2: Test results

Number of tests	Number of files	File size	Passed	Failed
20	50	2 MB	20	0
20	100	1 MB	20	0
20	20	5 MB	20	0
20	200	1 MB	20	0
20	200	2 MB	20	0
Sum	11400	18000 MB	100	0

Finally listing B.2 and listing B.3 show successful log files of the master and the slave node.

5.2 A File System for Safety-Critical Applications - tinysafefs

Tinysafefs is a wrapper file system which provides useful safety-related features for file systems that do not support them by them self. Tinysafefs does not claim to be ready for use in safety-critical systems, it is more a proof of concept for the extendibility of Linux and the whole GNU/Linux operating system.

Section 5.2.1 gives a short introduction to *FUSE* (Filesystem in Userspace), which provides a simple API for writing whole file systems in the so called *user space*. Section 5.2.2 will discuss the different modes that tinysafefs supports, section 5.2.3 will concentrate in the implementation of such a file system, section 5.2.4 provides information on compiling the file system, and finally section 5.2.5 shows a short test suite and its results.

5.2.1 FUSE

Tinysafefs is written with FUSE [9] which made the development fast and easy. According to the homepage, FUSE provides the following features:

- Simple library API
- Simple installation (no need to patch or recompile the kernel)
- Secure implementation
- Userspace - kernel interface is very efficient
- Usable by non privileged users
- Runs on Linux kernels 2.4.X and 2.6.X
- Has proven very stable over time

FUSE has a library part and a kernel module which communicate to each other via a device file (`/dev/fuse`). Figure 5.4 shows the structure of a file system call in FUSE.

5.2.2 Features

Tinysafefs has the following features which will be discussed in detail later in this section:

- Two disk mode with redundant writes and read checks
- TMR mode with fault masking and error correction
- Can be wrapped over different file systems as long as they are supported by Linux

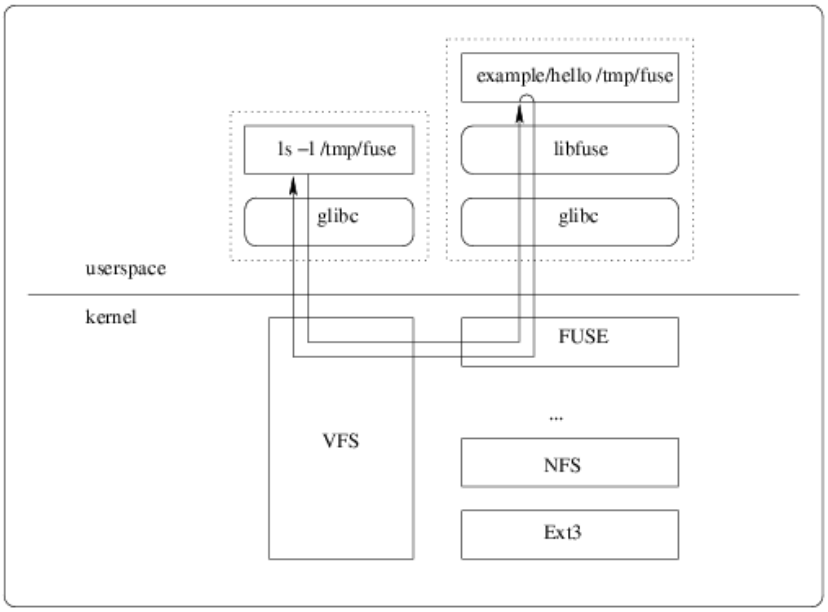

Figure 5.4: Structure of a system call in FUSE (http://fuse.sf.net)

In every mode tinysafefs is mounted to a directory (for example `/tmp/tinysafefs`) and utilizes two or three additional directories. To make sense, these directories should be mount points of additional physical disks or if wanted whole RAID arrays.

One major design decision was the decision between *checksums* or *redundant copies*. Whereas the approach with checksums has the major advantage that it does not cost as much space as writing files a second or even third time, there are advantages of the approach of writing files multiple times:

- Backups: Even if the file system cannot reconstruct data, maybe humans can do forensics.
- Only redundancy makes it possible to vote out and correct faulty data.
- To write a checksum of a file the whole file has to be read (compare `md5sum`), which would make a file system horrible slow. If checksum computation would be done in background this would compromise the safety arguments.
- Writing multiple times seams to be easier to implement in FUSE.

5.2.2.1 Two disk mode

In *two disk mode* every time data is written to the mount point of tinysafefs (`/tmp/tinysafefs`), it is mirrored to two directories, and therefore to two disks (which are mounted on the directories `/disk1` and `/disk2`). For example, if somebody creates a directory in `/tmp/tinysafefs`, the file system creates this directory on every physical disk. The following example demonstrates this behaviour. Note that tinysafefs is wrapped over `/tmp/tinysafefs` and the two disks are mounted on `/disk1` and `/disk2`.

```
$ cd /tmp/tinysafefs
$ ls
  testfile.txt
$ ls /disk*/
  /disk1:
  testfile.txt

  /disk2:
  testfile.txt
$ mkdir testdir
$ ls
  testfile.txt
  testdir
$ ls /disk*/
  /disk1:
  testfile.txt
  testdir

  /disk2:
  testfile.txt
  testdir
```

This example shows that data that is written to `/tmp/tinysafefs` is mirrored to both disks (`/disk1` and `/disk2`).

If data is read from `/tmp/tinysafefs`, it will be read from both disks and only if the two disks contain the same data, the file system will return this data back to the calling process. If the two disks contain different data an error will be raised. This adds *safety*, because a malicious disk cannot provide its malicious data. The following example shows how the file system reacts to non equal data. First, a file is created and read back. After that, one copy of the file gets destroyed and the file is read back again. This time the file system raises an error, because the files are not equal any more.

```
$ cd /tmp/tinysafefs
$ echo "testdata" > ./testfile.txt
$ ls /disk*
  /disk1:
  testfile.txt

  /disk2:
  testfile.txt
$ cat ./testfile.txt
  testdata
$ echo "destroy it" > /disk1/testfile.txt
$ cat ./testfile.txt
  cat: ./testfile.txt: No such file or directory
```

This behaviour adds safety because the application that reads data knows that something is wrong and can take action to prepare a safe state. For example if the direction of a switchblade

is encoded as a bit value where '0' means "left" and '1' means "right", a malicious hard disk that provides inverted data could cause serious damage. On the other hand, if the data is read with tinysafefs the application would read the error message and for example stop the train.

This behaviour of tinysafefs sounds like the reinvention of RAID-1, but this is not the case. RAID does not provide any safety features. RAID systems do not read the *whole* data from both disks. To improve read speed, parts of the file are read from one disk and other parts from an other disk. This behaviour of RAID systems does not improve safety aspects.

5.2.2.2 Three disk mode

In *three disk mode*, tinysafefs has the ability to tolerate one faulty disk and in the best case it can repair destroyed data. In every read cycle the date is read from three disks. After that, these read buffers get compared and if at least two of them are equal the data will be returned to the calling process. Tinysafefs uses TMR (Triple Modular Redundancy) and votes two out of three. If all three read buffers have a different view, an error will be raised. If tinysafefs works in correction mode, it tries to write back the correct data to the faulty file.

The following example shows tinysafefs in three disk mode with enabled correction.

```
1  $ cd /tmp/tinysafefs
2  $ echo "testdata" > ./testfile.txt
3  $ ls /disk*
4    /disk1:
5    testfile.txt
6
7    /disk2:
8    testfile.txt
9
10   /disk3:
11   testfile.txt
12 $ cat ./testfile.txt
13   testdata
14 $ echo "destroy it" > /disk1/testfile.txt
15 $ cat ./testfile.txt
16   testdata
17 $ cat /disk1/testfile.txt
18   testdata
19 $ echo "destroy it 1" > /disk1/testfile.txt
20 $ echo "destroy it 2" > /disk2/testfile.txt
21 $ cat ./testfile.txt
22   cat: ./testfile.txt: No such file or directory
```

The most important part is the second and third call of `cat`. The second one shows that one faulty file can be tolerated and the third one that the data gets corrected on `/disk1`.

5.2.3 Implementation

As stated before, *tinysafefs* is written with the help of FUSE (section 5.2.1). FUSE itself provides different programming language bindings. These bindings include *C*, *C++*, *perl*, *python*, and

several more. To make the file system as fast as possible, the choice to write it in *C* was obvious. To get the most out of the file system, the difference between *two disk mode* (section 5.2.2.1) and *three disk mode* (section 5.2.2.2) was done with C pre-processor `#define` constructs. On one hand this makes the code a bit more difficult to read, on the other hand this avoids additional CPU cycles for checking *if-then-else* statements. *If*-statements have to be checked every time, whereas `#define` statements are resolved during compile time by the pre-processor. For file systems it is better to get the most out of the kernel and the hardware and make the code a bit more difficult to read for the programmer.

FUSE provides a skeleton which maps file system calls to user defined functions, which is implemented via a *struct*. To get the idea a few example mappings are shown in listing 5.4.

Listing 5.4: Mapping of file system calls to functions

```
1  static struct fuse_operations tinysafe_oper = {
2      .getattr  = tinysafe_getattr,
3      .access   = tinysafe_access,
4      .readdir  = tinysafe_readdir,
5      .mknod    = tinysafe_mknod,
6      .mkdir    = tinysafe_mkdir,
7      .unlink   = tinysafe_unlink,
8      .rmdir    = tinysafe_rmdir,
9      .rename   = tinysafe_rename,
10     .chmod    = tinysafe_chmod,
11     .chown    = tinysafe_chown,
12     .truncate = tinysafe_truncate,
13     .open     = tinysafe_open,
14     .read     = tinysafe_read,
15     .write    = tinysafe_write,
16 };
```

In the rest of this section some of these functions will be discussed. Most of them are straight forward and follow the easy paradigm to do the ordinary operation a second or a third time to provide the former discussed redundancy. One of simplest functions is the one that is used to create new directories. The prototype and arguments of the functions are given by FUSE itself. The programmer's job is to fill this prototype with code that makes sense. Listing 5.5 shows the code for making redundant directory entries. Code enclosed between `#ifdef TMR` and `#endif` is executed if the file system is in *three disk mode*.

Listing 5.5: mkdir function of tinysafefs

```
1  static int tinysafe_mkdir(const char *path, mode_t mode)
2  {
3      int res;
4
5      res = mkdir(path, mode);
6      if (res == -1)
7          return -errno;
8
9      path += mount_dir_length;
10     GETPATH(backup_dir);
```

```
11
12      res = mkdir(sec_path, mode);
13      if (res == -1)
14          return -errno;
15
16  #ifdef TMR
17      GETPATH(backup_dir2);
18
19      res = mkdir(sec_path, mode);
20      if (res == -1)
21          return -errno;
22  #endif
23      return 0;
24  }
```

The macro *GETPATH* writes the backup paths (in the previous examples `/disk1` and `/disk2`) to the variable *sec_path* (second path), and addes the remaining file name from the original path variable. To keep the memory footprint small, *sec_path* is reused for the third directory operation. These are details and the most important thing to focus on is, that if there occurs a *single* `mkdir` call, behind the scene *multiple* directories are created.

Most of the really interesting code is in *tinysafe_getattr* (listing 5.6) and in *tinysafe_read*(listing 5.7 and 5.8).

The function `tinysafe_getattr` is called to get information about the file before it will be accessed. For example if a file is read, the getattr function will be called before the read function is called. This function sets a status buffer (*stbuf*), which will be used by FUSE internally. Therefore it is crucial that this status buffer is set to the correct values.

Listing 5.6: getattr function of tinysafefs

```
1  static int tinysafe_getattr(const char *path, struct stat *stbuf)
2  {
3      int res;
4  #ifdef TMR
5      struct stat sec_stbuf;
6      struct stat thrd_stbuf;
7  #endif
8
9      res = lstat(path, stbuf);
10     if (res == -1)
11         return -errno;
12
13 #ifdef TMR
14     if((S_ISREG(stbuf->st_mode)))
15     {
16         /* TMR check and vote for the right stat */
17         path += mount_dir_length;
18
19         GETPATH(backup_dir);
```

```
20          res = lstat(sec_path, &sec_stbuf);
21          if (res == -1)
22              return -errno;
23
24          GETPATH(backup_dir2);
25          res = lstat(sec_path, &thrd_stbuf);
26          if (res == -1)
27              return -errno;
28
29          /* TMR-vote */
30          if(stbuf->st_size == sec_stbuf.st_size)
31          {
32              if(sec_stbuf.st_size == thrd_stbuf.st_size)
33              {
34                  /* all equal */
35                  return 0;
36              }
37              else
38              {
39                  /* trunc third */
40                  GETPATH(backup_dir2);
41                  truncate(sec_path, stbuf->st_size);
42              }
43          }
44          else
45          {
46              if(sec_stbuf.st_size == thrd_stbuf.st_size)
47              {
48                  /* first faulty */
49                  path -= mount_dir_length;
50                  res = truncate(path, sec_stbuf.st_size);
51                  if(res == -1)
52                      return -errno;
53
54                  /* restat the path */
55                  res = lstat(path, stbuf);
56                  if (res == -1)
57                      return -errno;
58              }
59              else
60              {
61                  if(stbuf->st_size == thrd_stbuf.st_size)
62                  {
63                      /* trunc second */
64                      GETPATH(backup_dir);
65                      truncate(sec_path, stbuf->st_size);
66                  }
67                  else
```

```
68                  {
69                      return -ENOENT;
70                  }
71              }
72
73          }
74      }
75  #endif
76      return 0;
77  }
```

First of all the file systems tries to *stat* all the copies of the file. After that it checks which *stat structures* are equal and takes the necessary actions. If one of the backup copies is faulty the function sets it size to the correct one, which is done via *truncate*. If the fist copy is faulty the file system has to correct the size (*truncate*) and after this the *stat structure* has to be reread. After this is done, the file itself and the backup files are prepared for further actions.

The most important function is the read function because there faults are masked and repaired. This function is relatively long and therefore only the most important parts are shown in this section. If the file system is in *three disk mode*, three file descriptors are opened and the content of the file (with a given offset) is read to a buffer with *pread*. After that, all three buffers get compared and a variable *bufs_ok* is set according to the value of correct buffers. This variable can be seen as a bit field where every buffer has its own place an value. These values are combined with a *boolean or*. The following bit field shows the value of each buffer.

2	1	0	
three	two	one	} Buffers

For example, if buffers one and three are valid and the second one is faulty, *bufs_ok* would get the value *0x5*, which is `0x4 | 0x1` . The corresponding source code is shown in listing 5.7.

Listing 5.7: Comparison of buffers in the read function

```
1  static int tinysafe_read(const char *path, char *buf, size_t size,
2          off_t offset, struct fuse_file_info *fi)
3  {
4  ...
5  ...
6  ...
7      /* compare buffers */
8      if(memcmp(buf, sec_buf, size) != 0)
9      {
10         res = -ENOENT;
11     }
12 #ifdef TMR
13     else
14     {
15         bufs_ok |= (DISK0 | DISK1);
16         /* take one valid res als final result */
17         res = res_first;
```

```
18      }
19  #endif
20
21  #ifdef TMR
22      if(memcmp(buf, thrd_buf, size) == 0)
23      {
24          bufs_ok |= (DISK0 | DISK2);
25          res = res_first;
26      }
27      if(memcmp(sec_buf, thrd_buf, size) == 0)
28      {
29          bufs_ok |= (DISK1 | DISK2);
30          res = res_sec;
31      }
32  ...
33  ...
34  ...
35  #endif
36  ...
37  ...
38  ...
39  }
```

The next code snippet (listing 5.8) shows how files are corrected if tinysafefs is in *correction mode*. It is known which buffers contain the correct data (*bufs_ok*) and therefore correction is relatively easy. It is a *switch* statement which corrects the file that contains data that is not valid. The main focus is *safety*, not *availability*, therefore the file system does not raise an error if the file cannot be corrected.

Listing 5.8: Correction of files in the read function

```
1  ...
2  ...
3  ...
4      switch(bufs_ok)
5      {
6          case (DISK0 | DISK1): /* first, second ok, third broken */
7  #ifdef TMRC
8              /* write to third */
9              GETPATH(backup_dir2);
10             thrd_fd = open(sec_path, O_WRONLY);
11             if (thrd_fd != -1) /* try to correct */
12             {
13                 res_thrd = pwrite(thrd_fd, buf, res, offset);
14                 close(thrd_fd);
15             }
16 #endif
17             break;
18 ...
```

```
19 ...
20 ...
21     }
22 ...
23 ...
24 ...
```

So data is corrected while it is read. During testing the file system, one test was to copy a mp3 file to the file system and start playing it. During playback one copy was destroyed and the file system continued to return correct data because the faulty disk was voted out. During playback, the third faulty copy was reconstructed with the correct data from the first two disks. After playing back the first part of the mp3 file (before the destruction) the whole file on the third disk was corrected. This has been checked with the program `md5sum`.

One important thing to notice in figure 5.4 is, that FUSE communicates to the *VFS* (Virtual File System) which has a safety relevant advantage because someone gets diversity for free. If someone does not want to lay his or her trust on one singe file system it is possible to circumvent this problem with tinysafefs. Each disk can be formatted with a different file system as long as Linux supports it. For example the first disk could be formatted with *ext3*, the second one with *reiserfs*, and the third one with *xfs*.

5.2.4 Configuration

The steps to compile, mount, and unmount *tinysafefs* are described here. The first listing (listing 5.9) provides information on installing necessary tools for *tinysafefs* and the compilation of the package itself.

Listing 5.9: Intalling tools for tinysafefs and compilaton

```
1  $ sudo apt-get install pkg-config build-essential
2  $ sudo apt-get install fuse-utils libfuse-dev
3  $ mkdir ~/src
4  $ cd ~/src
5  $ # now copy the tarball to this directory before proceeding
6  $ ls
7    tinysafefs-0.0.1.tar.gz
8  $ tar -zxvf tinysafefs-0.0.1.tar.gz
9  $ cd tinysafefs-0.0.1
10 $ ./configure
11 $ make
```

This procedure compiles a binary called `tinysafefs`. This binary can be installed with `make install` or it can be used directly from this directory. Information on mounting is provided if the binary is called with `--help` as argument. An example mount is shown in listing 5.10.

Listing 5.10: Mounting tinysafefs

```
1 $ sudo modprobe fuse # load the fuse kernel module
2 $ mkdir /tmp/tinysafefs
3 $ # mount disks to /disk1, /disk2, /disk3 (not shown here).
```

```
$ # mount tinysafefs
$ sudo ./tinysafefs /tmp/tinysafefs --backup=/disk2 --backup2=/disk3 \
  -omodules=subdir,subdir=/disk1
$ # write test data to tinysafefs which will mirror the data
$ sudo dd if=/dev/urandom of=/tmp/tinysafefs/test.bin bs=1M count=10
$ # unmount file system
$ sudo fusermount -u /tmp/tinysafefs
```

That data written to `/tmp/tinysafefs` is mirrored to the backup directories can be shown with `ls -l /disk*`. To run further tests, the test suite which is shown in section 5.2.5 can be used.

5.2.5 Test Results

To test the functionality of the file system, a test suite written as a *Bash shell script* was developed and executed several times.

First the user of the suite has to set the variables in the suite according to the paths and has to set `VARSSET` to `YES`. If this configuration step is done the test suit can be executed as *root*. If it is executed as lower privileged user, the test suite aborts. After this check was successful the user can choose which tests should be executed.

The test functions are aware of the fact in which mode the file system is, because if the file system is in *two disk mode* the shell script variable `B2` is empty. The tests themselves are straight forward if basic concepts of GNU/Linux and the command line are known. In general, the test case creates a file and checks it with `md5sum`. Then the file is copied to the *tinysafefs* mount point and there it gets manipulated. For example it gets overwritten with other data. To verify this, the *md5sum* of the manipulated file is printed. After that, the file is reread from the file system and the *md5sum* is calculated. If the new checksum is equivalent to the first checksum, *tinysafefs* voted out the manipulated file. In all the other cases, the manipulated file is reset to the original state. In the case of the TMR mode with correction enabled, the test function does not reset the file, because the file has to be corrected by the file system itself. To test if the file system responds with an error if all copies of the file are destroyed there is a test case which destroys every single copy of the file. In this case *tinysafefs* responds with an error.

Listing B.4 shows a successful run of the test suite and the entries in the `/var/log/syslog`. As it can be seen in the log file, the destroyed file was voted out, and because the file system was in TMR mode *with correction*, the file was corrected on the read access.

Table 5.3 shows the test results for *tinysafefs*.

Mode	Number of tests	Passed	Failed
two disk	50	50	0
TMR	50	50	0
TMRC	50	50	0
Sum	150	150	0

Table 5.3: Test results

6 Conclusion

The discussion about the possibility of using GNU/Linux in safety-critical applications will be resumed and immersed in the following part of the thesis. The goal is to connect the theoretical parts of the previous chapters and make a statement relevant for practical, real life safety-critical systems.

The thesis started with an introduction (chapter 1), that gave a description of the problem. In the modern world we live nowadays, mankind depends on a variety of technical, highly sophisticated systems. To guarantee the *safety* of the environment and the safety of human beings, there are claims that these systems have to fulfil. It is necessary that there is an independent layer that guarantees that a system is safe, which is provided by *standards*. There are several companies that provide special software for such safety-critical systems, but there is an alternative to this proprietary solutions, which is the operating system called *GNU/Linux*. This operating system, and its kernel Linux, is well known and used in thousands of desktop computers, servers and embedded systems all over the world. Therefore, it is a natural question, if this operating system, or parts of it, can be used for safety-critical applications and systems. As there are different opinions on this issue the rest of the thesis dealt with this question.

To give an answer to the question if GNU/Linux and Linux can be used for safety-critical systems some fundamental terms had to be clarified.

Chapter 2 dealt with the kernel Linux and the GNU/Linux operating system. This chapter started with an overview what an operating system kernel is and what it has to do. It is the component of an operating system, that is between the hardware and the applications that want to communicate to this hardware. After providing a short overview about the history and the features of Linux, the development of the kernel was examined. It is significant for the evidence that a software component is ready for use in safety-critical systems, to trust the way it is developed. Therefore different development models were compared and the chapter showed that the kernel community uses sophisticated tools (section 2.4.3) for the development. The development team has a well defined structure and hierarchy. Besides from that it follows a release cycle (section 2.4.4), that helps to provide stable software.

Chapter 3 concentrated on the basic terms that are used for safety-critical systems. This declarations were necessary to provide a stable base what the term *safety* means and to trace the outline to terms like *reliability*, *availability*, and *maintainability* (section 3.2). The next section of this chapter concentrated on an essential topic for safety-critical systems, which was *testing* (section 3.3). There the differences between several testing methods like *black-box testing*, *white-box testing*, and *fault injection* were outlined. After that, the focus was laid on safety-related *standards*.

Section 3.4 discussed the differences between several standards and showed the requirements that a system has to fulfil to get certified according to these standards. As GNU/Linux distributions are components of the shelf, the discussion if *COTS* can be used for safety-critical systems was given in section 3.5.1. After lining out what the term COTS stands for, an overview about existing GNU/Linux components of the shelf was given in section 3.5.2.

After giving all the basics, chapter 4 discussed several related papers that deal with GNU/Linux in safety-critical systems and COTS components for safety-critical systems. The main discussion was laid out on two very sceptical papers, but it was shown that most of the arguments can be extenuated or even disproved. Some arguments simply do not hold any more, because the GNU/Linux, and especially the Linux kernel community made a huge progress in the right direction. Some other arguments can be seen from another point of view, which lays much more trust in the Open Source development model.

Chapter 5 demonstrated that it is easy to contribute to the GNU/Linux community and to help it forward to become ready for safety-critical systems. The chapter contains two parts, where the fist (section 5.1) tests RAID-1 systems via a client/server architecture test suite. With the addition of a simple hardware component and a Linux kernel module it is possible to test RAID-1 systems in a realistic way. The second part (section 5.2) provided an extension for already existing file systems. The *tinysafefs* adds a layer of safety that can be used directly by file systems on hard disks or as a meta file system for RAID-1 systems. Both projects have shown that it is relatively easy to test and to extend Linux kernel components. This showed that it is much easier to improve FOSS components than proprietary ones.

As outlined in chapter 3 (section 3.4) a software component has to conform to safety-related *standards* to be relevant for practical use. These standards have to be rigorous to ensure the safety of the environment and the safety of human beings. Therefore, they force rules for developing safety-critical software and safety-critical systems. Most of the standards seem to be written for a traditional software developing approach, but on the other hand their terminology is generic enough that someone certifying a component according to theses standards is not limited to one unique approach. For example the standard EN 50129 [CEN03] contains phrases like “evidence of quality management”, but it does not force a unique technique to achieve this goal. This is an excellent approach which makes it possible to certify a component that was developed using an unusual way.

To be successful, the Open Source community had to create new ways of working together from all over the world. These methods are new and may seem to be unconventional, but they work very well in practice. For example, *git*, the source code management tool of Linux, is a new and extraordinary approach because it breaks with the traditional rules of centralized code repositories and follows a decentralized design. The increased quality of the kernel since new approaches have been taken, not only *git* itself, is an indication that quality management can be achieved with new approaches. *Git* is just one little example which demonstrates the predominance of Free and Open Source development. This can be seen from the lowest level, for example code management, up to the highest level of GNU/Linux distributions. The aim of these examples is to show that there is a major line in argumentation that connects Free and Open Source to the main question in of this theses: “Is it possible to use GNU/Linux (and FOSS) in a safety-related context?” The common thread that runs through the thesis is that Free and Open Source use different, high sophisticated approaches to fulfil traditional goals. As it has been shown, the kernel community does this very well, in most cases better than the traditional approaches. The FOSS development model itself does not claim that it bears software that can be used for safety-critical systems, but

FOSS definitely supports the argumentation that software developed according to this model has a higher chance to be major enough to be used for safety-critical applications. This is shown in the practical part (chapter 5) of this thesis. If things are not adequate, just make them adequate. Even if the tools and extensions presented in chapter 5 are not enough, and not enough tested to be used in real world safety-critical systems, they show the strength of Free and Open Source software.

One important thing to notice is that FOSS is not the holy grail, but it can be used for safety-critical applications, even for applications that have a high safety integrity level. FOSS is not a limiting factor for safety, it is a benefit. For example, this was shown in section 3.3.2 when white-box testing was discussed. FOSS makes it possible to do white-box testing, because the source is available. Of course, taking a broken piece of software and make it open source does not mean that it will become better over night, but if there are developers interested in this piece of software, the quality will improve. There are major companies and organisations that are very active in pushing Linux in the right direction (for example OSADL [33]).

It is important to know what *certification* really means, which is different in several standards, and what consequences it brings. For example if the question is "Can a stock Linux kernel itself be certified according to standard XYZ?", then in most cases the answer is a clear "no", because it is not possible to do a line by line testing of such a huge component as the Linux kernel. The more interesting question is: "Is it possible to use Linux in a safety-relevant context?" There the answer is different, and often it is "yes". There are cases where Linux can be used. Depending on the project, the use of a flattened version of the kernel that does not contain unused subsystems should be considered. It is important to notice that in several standards a *whole system* is certified and not a single component. If this is the case, the kernel is not the only part that is responsible for the whole system. There are measures that can be taken to guarantee that the system is safe even if there are small shortcomings in a single component. For example, diversity can be used, or a small layer that checks the results of the underlying system. To conclude this example, a safety layer could be used to check the results and the timing of the underlying Linux kernel. If the Linux kernel takes too long to response the system is turned to a *safe state*.

Appendices

A Extreme Programming

Table A.1: Extreme programming values according to Kent Beck [Bec05]

Value	Description
Communication	Every software development process needs communication between the developers. In the former named traditional models this communication is done by *documentation*. The output of one formal step is used as input for the next step. Extreme Programming has another paradigm that tries to force communication. The method in which this communication is done is secondarily. For example feedback via verbal communication. This idea of fast and frequent communication can be found in the Open Source model, too, where communication is done via mailing lists or IRC-channels.
Simplicity	Try to keep the code and the design simple. Some traditional models force an overall view of the system and software design before the coding is done. Extreme programming tries to solve currently known problems and not the problems that might come in the future. The advantage is that there is no time investigated for future problems that might never appear because of the rapid requirement changes. This near-sighted view has the disadvantage that it is sometimes more work to change the design at a later point in time than design it from the beginning in a sufficient way. This idea of *simplicity* can also be found in the Open Source development model. It is usual to release a core component and let the community develop on this base. It is the old "release often and early" paradigm that works very well, it is a new evolutionary process of software development.
Feedback	The term feedback is divided into several sub categories. *From the system* by testing, *from the customer* by integration tests and *from the team*.

Courage	This includes design and coding decisions. For the design part it means that developers should only develop for today and not for tomorrow. Coding often needs courage, too, because sometimes it is necessary to throw away portions of code. Developers should not be afraid to do that for wrong reasons. For example it is hard to throw away code which needed a long time to be implemented, but if it is the right decision it has to be done.
Respect	Respect means that the developers should respect other developers and their work. This is also important for Open Source development where developers do not personally know their counterparts. For example it is a bad idea to patch a system in such a way that it helps the sub system you are working on while it breaks another sub system.

Table A.2: Extreme programming practices according to Kent Beck [Bec05]

Practice	Description
Fine scale feedback	
Pair programming	The idea of pair programming is not new, it is one of the oldest ideas in software development. For example it was used in times when computers costed much more money and not every programmer had his or her own terminal. Pair programming is a process of *coding* and *watching*. One programmer is focused on coding while the second one is focused on the overall picture.
Planing Game	This term describes the future of the project and is divided in two main phases. The first one is *Release Planning*, where the goals of the next release are planed and *Iteration Planning*. In release planning the customer of the software is included while iteration planning is for the programmers only.
Test driven development	In test driven development so called *unit tests* [Boa99] are used to test the correctness of the software. It is important to mention that these unit tests are coded before the implementation of the function under test is undertaken.
Whole team	This practice means that the whole team is included in the development process. This team also includes the customer that should be available for all kinds of questions during the development. It makes the customer a member of the team and the customer is not only that one that pays the cheque. It is interesting that this idea is very common for open source development, too. Except from business driven development the usual case is that the customer is the programmer of the software. In this case the customer can be a whole community that is included.
Continuous process	

Continuous Integration	When many developers work on a project it is important that they commit their changes to some kind of software repository. If they wait to long integration problems may appear. Continuous integration prevents this issue and all developers have the view of working in the same version. This practice can be seen as the open source counterpart of "release often and early".
Design Improvement	Extreme programming forces the so called *KISS* (keep it simple, stupid) paradigm which may lead to that point where the development sticks. If that occurs it is important to reconsider the design.
Small Releases	Small releases should be done in a continuous way. This helps the development in general and supports the *continuous integration* practise and it helps the customer, too. The customer has the ability to get confirmed that the development makes progress and therefore the customer can give feedback which supports the *whole team* practice.
Shared understanding	
Coding standards	Coding standards should be used and accepted by the whole development team. Coding standards are necessary to make collaboration between different programmers much easier.
Collective Code Ownership	This principle means that every developer should know the whole code and should be able to fix bugs in the code. Therefore every developer is responsible for the quality of the product, not only for his or her part of the project.
Simple Design	Always use the simplest solution for a problem. If there is a simpler solution for a function already implemented nobody should hesitate to rewrite and refactor this part of the code.
System Metaphor	This discipline describes a naming convention for functions and classes in the system. It should be easy to guess the function of a class by its name. For example a function that appends text to a log file could have the name "append_to_log()"
Programmer welfare	
Sustainable Pace	This discipline says that a programmer should not be exploited, a sustainable pace is much better for the progress of the system under development. If someone is forced to work too long on a problem he or she gets frustrated and drained.

B Selected Source Code

B.1 raid1test

Listing B.1: communication.h

```
1  /*
2   * Copyright (C) 2008 - Roland Kammerer
3   * This program is free software; you can redistribute it and/or
4   * modify it under the terms of the GNU General Public License
5   * as published by the Free Software Foundation; either version 2
6   * of the License, or (at your option) any later version.
7   *
8   * This program is distributed in the hope that it will be useful,
9   * but WITHOUT ANY WARRANTY; without even the implied warranty of
10  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11  * GNU General Public License for more details.
12  *
13  * You should have received a copy of the GNU General Public License
14  * along with this program; if not, write to the Free Software
15  * Foundation, Inc., 59 Temple Place - Suite 330, Boston,
16  * MA  02111-1307, USA.
17  */
18
19 /**
20  * @file communication.h
21  * @brief Functions for socket communication.
22  * @author Roland Kammerer
23  * @date 2008-09-24
24  *
25  * This library provides basic functions for socket communication.
26  * This includes the creation of a socket, initialisation of a socket
27  * address and accepting new connections. The library provides
28  * functions to send and receive messages to/from sockets.
29  * Message logging to stdout/stderr or files is done here, too.
30  */
31
```

```
32 #ifndef _COMMUNICATION_H
33 #define _COMMUNICATION_H
34
35 #define STDOUTMSG "-"
36
37 /** @brief Create a new socket */
38 int make_socket(uint16_t port);
39
40 /** @brief Accepts new connection to a socket. */
41 int accept_new_connection(int sockfd, struct sockaddr_in *cli_addr,
42                           int clilen);
43
44 /** @brief Initialize a socket address before we connect to the server
   */
45 void init_sockaddr(struct sockaddr_in *name, const char *hostname,
46                    uint16_t port);
47
48 /** @brief Send a message using a socket. */
49 int send_message(char *msg, int sockfd);
50
51 /** @brief Read a message from a socket and store it to a buffer. */
52 int read_message(char *buffer, int sockfd);
53
54 /** @brief Write a messge to a logfile/stdout/stderr. */
55 int log_message(char *writeto, char *message);
56
57 /** @brief Read a message from a socket, store it to a buffer
58  * and write it to a logfile/stdout/stderr. */
59 int read_and_log(char *buffer, int sockfd, char *writeto);
60 #endif
```

Listing B.2: Logfile of the master node

```
1 --START: Mon Aug 11 13:30:14 2008
2
3 Starting ./raid1testmaster on port 1234
4
5 SLAVE: initialized
6 SLAVE: Ok
7 f6b87ebeb9aabf36eb8efab18da230432989021e
8 SLAVE: Test 1 successful
9 SLAVE: Turn disk0 off
10 press any key
11 SLAVE: Ok
12 f6b87ebeb9aabf36eb8efab18da230432989021e
13 SLAVE: Ok
14 4817baa6976ae12370969e26a7008ec2b8a5fcee
15 SLAVE: Turn disk0 on
16 press any key
17 SLAVE: Test 2 successful
18 SLAVE: initialized
```

```
SLAVE: Ok
4817baa6976ae12370969e26a7008ec2b8a5fcee
SLAVE: Ok
4817baa6976ae12370969e26a7008ec2b8a5fcee
SLAVE: Test 3 successful
SLAVE: Ok
fd5528241bb5792a8b277344581ff1d7520d0584
SLAVE: Test 4 successful
SLAVE: Turn disk1 off
press any key
SLAVE: Ok
fd5528241bb5792a8b277344581ff1d7520d0584
SLAVE: Ok
fa34952e04ae94493de724c60d740b72f943f14b
SLAVE: Turn disk1 on
press any key
SLAVE: Test 5 successful
SLAVE: initialized
SLAVE: Ok
fa34952e04ae94493de724c60d740b72f943f14b
SLAVE: Ok
fa34952e04ae94493de724c60d740b72f943f14b
SLAVE: Test 6 successful
SLAVE: Exit
All tests ran successful!
---END: Mon Aug 11 13:54:37 2008
```

Listing B.3: Logfile of the slave node

```
Binding /usr/sbin/raid1testslave to Server masternode on port 1234

MASTER: Run Test 1
---START: Mon Aug 11 13:31:13 2008

          do testcase 1
generating files...
checksum_write: f6b87ebeb9aabf36eb8efab18da230432989021e
reading files...
checksum_read: f6b87ebeb9aabf36eb8efab18da230432989021e
MASTER: Get checksum
MASTER: Got checksum
MASTER: Run Test 2
          do testcase 2
MASTER: Switched disk0 off
mdadm --detail /dev/md1 1>/dev/null
checking new status
reading files...
reading files successfull...
MASTER: Get checksum
MASTER: Got checksum
MASTER: Checksum ok
generating files...
checksum_write: 4817baa6976ae12370969e26a7008ec2b8a5fcee
reading files...
checksum_read: 4817baa6976ae12370969e26a7008ec2b8a5fcee
```

```
27 MASTER: Get checksum
28 MASTER: Got checksum
29 umount /dev/md1
30
31 MASTER: Switched disk0 on
32 MASTER: Reboot 1
33 rebooting...
34 Binding /usr/sbin/raid1testslave to Server lapwlan on port 1234
35
36 MASTER: Run Test 3
37          do testcase 3
38 checking status
39 reading files...
40 reading files successfull...
41 MASTER: Get checksum
42 MASTER: Got checksum
43 MASTER: Checksum ok
44 mdadm /dev/md1 --add /dev/firstraiddisk
45
46 waiting for sync...
47 approximately synced in 706 min; 90 sec
48         recheck in 30 seconds
49 approximately synced in 3 min; 70 sec
50         recheck in 30 seconds
51 approximately synced in 3 min; 0 sec
52         recheck in 30 seconds
53 approximately synced in 2 min; 50 sec
54         recheck in 30 seconds
55 approximately synced in 2 min; 10 sec
56         recheck in 30 seconds
57 approximately synced in 1 min; 90 sec
58         recheck in 30 seconds
59 approximately synced in 1 min; 30 sec
60         recheck in 30 seconds
61 approximately synced in 0 min; 90 sec
62         recheck in 30 seconds
63 approximately synced in 0 min; 60 sec
64         recheck in 30 seconds
65 approximately synced in 0 min; 20 sec
66         recheck in 30 seconds
67 RAID synced
68 reading files...
69 reading files successfull...
70 MASTER: Get checksum
71 MASTER: Got checksum
72 MASTER: Checksum ok
73 MASTER: Run Test 4
74          do testcase 4
75 generating files...
76 checksum_write: fd5528241bb5792a8b277344581ff1d7520d0584
77 reading files...
78 checksum_read: fd5528241bb5792a8b277344581ff1d7520d0584
79 MASTER: Get checksum
80 MASTER: Got checksum
```

```
81 MASTER: Run Test 5
82          do testcase 5
83 MASTER: Switched disk1 off
84 mdadm --detail /dev/md1 1>/dev/null
85 checking new status
86 reading files...
87 reading files successfull...
88 MASTER: Get checksum
89 MASTER: Got checksum
90 MASTER: Checksum ok
91 generating files...
92 checksum_write: fa34952e04ae94493de724c60d740b72f943f14b
93 reading files...
94 checksum_read: fa34952e04ae94493de724c60d740b72f943f14b
95 MASTER: Get checksum
96 MASTER: Got checksum
97 umount /dev/md1
98
99 MASTER: Switched disk1 on
100 MASTER: Reboot 1
101 rebooting...
102 Binding /usr/sbin/raid1testslave to Server lapwlan on port 1234
103
104 MASTER: Run Test 6
105          do testcase 6
106 checking status
107 reading files...
108 reading files successfull...
109 MASTER: Get checksum
110 MASTER: Got checksum
111 MASTER: Checksum ok
112 mdadm /dev/md1 --add /dev/secondraiddisk
113
114 waiting for sync...
115 approximately synced in 706 min; 90 sec
116         recheck in 30 seconds
117 approximately synced in 8 min; 40 sec
118         recheck in 30 seconds
119 approximately synced in 7 min; 50 sec
120         recheck in 30 seconds
121 approximately synced in 6 min; 90 sec
122         recheck in 30 seconds
123 approximately synced in 6 min; 20 sec
124         recheck in 30 seconds
125 approximately synced in 6 min; 0 sec
126         recheck in 30 seconds
127 approximately synced in 5 min; 60 sec
128         recheck in 30 seconds
129 approximately synced in 5 min; 10 sec
130         recheck in 30 seconds
131 approximately synced in 4 min; 60 sec
132         recheck in 30 seconds
133 approximately synced in 4 min; 20 sec
134         recheck in 30 seconds
```

```
135 approximately synced in 3 min; 80 sec
136         recheck in 30 seconds
137 approximately synced in 3 min; 40 sec
138         recheck in 30 seconds
139 approximately synced in 2 min; 90 sec
140         recheck in 30 seconds
141 approximately synced in 2 min; 40 sec
142         recheck in 30 seconds
143 approximately synced in 1 min; 90 sec
144         recheck in 30 seconds
145 approximately synced in 1 min; 60 sec
146         recheck in 30 seconds
147 approximately synced in 1 min; 10 sec
148         recheck in 30 seconds
149 approximately synced in 0 min; 60 sec
150         recheck in 30 seconds
151 approximately synced in 0 min; 10 sec
152         recheck in 30 seconds
153 RAID synced
154 reading files...
155 reading files successfull...
156 MASTER: Get checksum
157 MASTER: Got checksum
158 MASTER: Checksum ok
159 MASTER: Run Test -1
160 Got EXIT message from Master
161 ---END: Mon Aug 11 13:54:37 2008
```

B.2 tinysafefs

Listing B.4: Syslog output of tinysafefs

```
1  Aug  9 15:27:52 ubuntu tinysafefs[4408]: Voted out /disk1
2  Aug  9 15:27:52 ubuntu tinysafefs[4408]: Corrected /disk1/filetest.bin
3  Aug  9 15:27:57 ubuntu tinysafefs[4408]: Voted out /disk2
4  Aug  9 15:27:57 ubuntu tinysafefs[4408]: Corrected /disk2/filetest.bin
5  Aug  9 15:28:01 ubuntu tinysafefs[4408]: Voted out /disk3
6  Aug  9 15:28:01 ubuntu tinysafefs[4408]: Corrected /disk3/filetest.bin
7  Aug  9 15:28:05 ubuntu tinysafefs[4408]: All disks inconsistent (Rack on fire!)
8  Aug  9 15:28:17 ubuntu tinysafefs[4408]: Voted out /disk1
9  Aug  9 15:28:17 ubuntu tinysafefs[4408]: Corrected /disk1/test.txt
10 Aug  9 15:28:18 ubuntu tinysafefs[4408]: Voted out /disk2
11 Aug  9 15:28:18 ubuntu tinysafefs[4408]: Corrected /disk2/test.txt
12 Aug  9 15:28:19 ubuntu tinysafefs[4408]: Voted out /disk3
13 Aug  9 15:28:19 ubuntu tinysafefs[4408]: Corrected /disk3/test.txt
14 Aug  9 15:28:20 ubuntu tinysafefs[4408]: All disks inconsistent (Rack on fire!)
```

Literature

[AALC92] Dimiter R. Avresky, Jean Arlat, Jean-Claude Laprie, and Yves Crouzet. Fault Injection for the Formal Testing of Fault Tolerance. In *FTCS*, pages 345–354, 1992.

[ALR01] A. Avizienis, J. Laprie, and B. Randell. Fundamental Concepts of Dependability. Technical report, University of California, Los Angeles, LAAS-CNRS Toulouse, France, University of Newcastle upon Tyne, U.K., April 2001.

[ARI92] ARINC. Software Considerations in Airborne Systems and Equipment Certification. Technical report, ARINC, Annapolis, Maryland, 1992.

[BA99] Barry Boehm and Chris Abts. COTS Integration: Plug and Pray? *Computer*, 32(1):135–138, 1999.

[Bal08] Helmut Balzert. *Lehrbuch der Softwaretechnik: Softwaremanagement.* Spektrum Akademischer Verlag, 2008.

[Bau07] Alexander Bauer. Realtime capabilities of low-end PowerPC and ARM boards for embedded systems. In *Proceedings of the Ninth Real-Time Linux Workshop in Linz*, November 2007.

[BBD+00] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, April 2000.

[Bec05] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman, Amsterdam, 2005.

[Boa99] IEEE Standards Board. IEEE Standard of Software Unit Testing Std 1008-1987. Technical report, IEEE, 1999.

[Boe88] Barry Boehm. A Spiral Model of Software Development and Enhancement. In *Proceedings of IEEE*, pages 61–72, May 1988.

[BW06] Martin Bligh and Andy P. Whitcroft. Fully Automated Testing of the Linux Kernel. In *Proceedings of the Linux Symposium*, pages 113–126, 2006.

[CA78] Liming Chen and Algirdas Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing 1995, Highlights from Twenty-Five Years.* FTCS, 1978.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (ICSS)*. Addison Wesley, 1994.

[CEN99] CENELEC. Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Technical report, European Committee for Electrotechnical Standardization, 1999.

[CEN00] CENELEC. Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems. Technical report, European Committee for Electrotechnical Standardization, November 2000.

[CEN03] CENELEC. Railway applications - Communications, signalling and processing systems - Safety related electronic systems for signalling. Technical report, European Committee for Electrotechnical Standardization, 2003.

[Dah72] Ole-Johan Dahl. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[Dik06] Jeff Dike. *User Mode Linux (Bruce Perens' Open Source Series)*. Prentice Hall, 2006.

[DK97] Stefan Dawkins and Tim Kelly. Supporting the use of COTS in safety critical applications. *Cots and Safety Critical Systems (Digest No. 1997/013), IEE Colloquium on*, pages 8/1–8/4, Jan 1997.

[FG] Rainer Faller and William M. Goble. Open IEC 61508 Certification of Products. Technical report, Exida.com.

[FM95] Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle. Technical report, Center for Systems Management, 19046 Pruneridge Avenue, Cupertino, CA 95014, 1995.

[Gäb06] Rene Gäbler. *Dreambox kompakt*. Bomots Verlag, 2006.

[Gre01] Grechenig. *Software Engineering*. Pearson Studium, 2001.

[Gro04] Real-Time Systems Group. LU Distributed Real-Time Systems Engineering. Technical report, Institute of Computer Engineering, 2004.

[Gui07] Nicholas Mc Guire. Linux for Safety Critical Systems in IEC 61508 Context. In *Proceedings of the Ninth Real-Time Linux Workshop in Linz*, November 2007.

[Het88] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[HWS04] H. Höxner, Martin Waitz, and V. Sieh. Advanced virtualization techniques for FAUmachine. *Linux Kongress*, pages 1–12, September 2004.

[IEC98] IEC. Functional Safety of electrical/electronic/programmable electronic safety-related systems. Technical report, International Electrotechnical Commission, 1998.

[JBFB01] C. Jones, R. E. Bloomfield, P. K. D. Froome, and P. G. Bishop. Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP). Technical report, Adelard for the Health and Safety Executive, 2001.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. In *IEEE Special Issue on Modeling and Design of Embedded Software*, pages 112–126, 2003.

[KG93] Hermann Kopetz and G. Grünsteidl. TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 524–533, 1993.

[KK95] Heinz Kantz and Christian Koza. The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity. In *FTCS '95: Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 453–458, Washington, DC, USA, 1995. IEEE Computer Society.

[Kop97] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications (Kluwer International Series in Engineering & Computer Science)*. Springer Netherlands, 1997.

[Kra05] Martin Krafft. *The Debian System: Concepts and Techniques*. No Starch Press, 2005.

[Kri02] Linda Kristiansen. COTS components in safety-critical systems. Master's thesis, The Norwegian University for Science and Technology (NTNU), June 2002.

[Lap91] J. C. Laprie. *Dependability: Basic Concepts and Terminology : In English, French, German, Italian and Japanese (Dependable Computing and Fault-Tolerant Systems)*. Springer-Verlag, 1991.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[Lio96] J. L. Lions. ARIANE 5: Flight 501, Report by the Inquiry Board. Technical report, European Space Agency, 7 1996.

[MSR07] Morten Mossige, Pradyumna Sampath, and Rachana Rao. Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power technologies - A case study. In *Proceedings of the Ninth Real-Time Linux Workshop in Linz*, November 2007.

[MT01] Maurizio Morisio and Marco Torchiano. Definition and Classification of COTS: A proposal. *Lecture Notes in Computer Science*, 2255:165–175, 2001.

[Mye79] Glenford J. Myers. *The Art of Software Testing (Business Data Processing)*. John Wiley and Sons, 1979.

[PB93] Gustav Pomberger and Günther Blaschek. *Grundlagen des Software - Engineering. Prototyping- und objektorientierte Software- Entwicklung*. Hanser Fachbuchverlag, 1993.

[Pie02] Ron Pierce. *Preliminary Assessment of Linux for Safety Related Systems (Research Report)*. Health and Safety Executive (HSE), 8 2002.

[POS04] POSIX. *POSIX.1/IEEE 1003.1-2001*. POSIX, 2004.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[RH07] Steven Rostedt and Darren V. Hart. Internals of the RT Patch. In *Proceedings of the Ottawa Linux Symbosium*, pages 161–172, June 2007.

[Roy70] Winston Royce. Managing the Development in Large Software Systems. In *Proceedings of IEEE WESCOM*, 1970.

[Rus99] John Rushby. Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[Rus07] Rusty Russell. lguest: Implementing the little Linux Hypervisor. In *Proceedings of the Ottawa Linux Symbosium*, pages 173–178, June 2007.

[Sta97] Victoria Stavridou. COTS, Integration and Critical Systems. Technical report, Department of Computer Science, Queen Mary and Westfield Colleage, 1997.

[Sta02] Michael Stamatelatos. Fault Tree Handbook with Aerospace Applications. Technical report, NASA, U.S.A., August 2002.

[Str80] Lee Patrick Strobel. *Reckless Homicide? Ford's Pinto Trial.* And Books, 1980.

[Tal98] Nancy Talbert. The Cost of COTS. *IEEE Computer*, 31(6):46–52, 1998.

[Tan03] Tanenbaum. *Verteilte Systeme.* Pearson Studium, 2003.

[TD02] Linus Torvalds and David Diamond. *Just For Fun: The Story of an Accidental Revolutionary.* Texere Publishing,US, 2002.

[Voa98] Jeffrey Voas. COTS: The Economical Choice? *IEEE Software*, 15(3):16–19, March,April 1998.

[Wie98] Michael Wielsch. *Linux.* Data Becker GmbH + Co.Kg, 1998.

Internet References

[1] An introduction to RTLinux. *Homepage*, September 2008. http://www.linuxdevices.com/articles/AT3694406595.html.

[2] Anatomy of the Linux kernel. *Homepage*, September 2008. http://www.ibm.com/developerworks/linux/library/l-linux-kernel/.

[3] Article about the real-time capabilities of Linux. *Homepage*, September 2008. http://edageek.com/2006/10/12/real-time-linux/.

[4] Bugzilla - Bug reporting tool. *Homepage*, September 2008. http://www.bugzilla.org.

[5] The Case of the Ford Pinto. *Homepage*, September 2008. http://www.cs.rice.edu/~vardi/comp601/case2.html.

[6] Debian GNU/Linux. *Homepage*, September 2008. http://www.debian.org.

[7] Doxygen - Source code documentation. *Homepage*, September 2008. http://www.doxygen.org.

[8] Dreambox - Digital Video Recoder. *Homepage*, September 2008. http://www.dream-multimedia-tv.de.

[9] Filesystem in Userspace. *Homepage*, September 2008. http://fuse.soureforge.net.

[10] The FreeBSD Operating System. *Homepage*, September 2008. http://www.freebsd.org.

[11] FreeBSD Release Process. *Homepage*, September 2008. http://www.freebsd.org/doc/en_US.ISO8859-1/articles/releng/release-proc.html.

[12] GNU debugger. *Homepage*, September 2008. http://sourceware.org/gdb.

[13] GNU/Hurd kernel. *Homepage*, September 2008. http://www.gnu.org/software/hurd/hurd.html.

[14] GNU/Linux bases Linksys routing device. *Homepage*, September 2008. http://en.wikipedia.org/wiki/Wrt54gl.

[15] GNU's Not UNIX. *Homepage*, September 2008. www.gnu.org.

[16] Gobuntu GNU/Linux. *Homepage*, September 2008. http://www.ubuntu.com/products/whatisubuntu/gobuntu.

[17] Gobuntu wiki. *Homepage*, September 2008. http://wiki.ubuntu.com/Gobuntu.

[18] GPG - Gnu Privacy Guard. *Homepage*, September 2008. http://www.gnupg.org.

[19] GPL compatible Licenses. *Homepage*, September 2008. http://www.gnu.org/licenses/license-list.html#SoftwareLicenses.

[20] How to report Linux kernel bugs. *Homepage*, September 2008. http://kernel.org/pub/linux/docs/lkml/reporting-bugs.html.

[21] InfiniBand Trade Association. *Homepage*, September 2008. http://www.infinibandta.org.

[22] Kernel Performance Testing. *Homepage*, September 2008. http://kernel-perf.sourceforge.net.

[23] Kernel Testing. *Homepage*, September 2008. http://test.kernel.org.

[24] Launchpad - Free software hosting and development website. *Homepage*, September 2008. https://launchpad.net.

[25] Lguest - Kernel Virtualisation. *Homepage*, September 2008. http://lguest.ozlabs.org.

[26] Linux Foundation. *Homepage*, September 2008. http://www.linux-foundation.org/en/About.

[27] The Linux Kernel Archives. *Homepage*, September 2008. http://www.kernel.org.

[28] The Linux Kernel Mailing List Archive. *Homepage*, September 2008. http://www.lkml.org.

[29] Linux Kernel Source Level Debugger. *Homepage*, September 2008. http://kgdb.linsyssoft.com.

[30] Linux Test Project. *Homepage*, September 2008. http://ltp.sourceforge.net.

[31] Linux v2.6 scales the enterprise. *Homepage*, September 2008. http://www.infoworld.com/infoworld/article/04/01/30/05FElinux_1.html.

[32] LTP - RAID1 test plan. *Homepage*, September 2008. http://ltp.sourceforge.net/documentation/test_plans/raid1.php.

[33] Open Source Development Labs. *Homepage*, September 2008. http://www.osadl.org.

[34] Open Source Initiative. *Homepage*, September 2008. http://www.opensource.org.

[35] Overview of the different kinds of Ubuntu developers. *Homepage*, September 2008. http://wiki.ubuntu.com/UbuntuDevelopers.

[36] Real-Time Linux Wiki. *Homepage*, September 2008. http://rt.wiki.kernel.org.

[37] Red Hat - GNU/Linux distribution. *Homepage*, September 2008. http://www.redhat.com.

[38] Red Hat Enterprise Linux details. *Homepage*, September 2008. http://www.redhat.com/rhel/server/details.

[39] Red Hat Enterprise Linux Errata Support Policy. *Homepage*, September 2008. http://www.redhat.com/security/updates/errata.

[40] Red Hat MRG. *Homepage*, September 2008. http://www.redhat.com/mrg/realtime.

[41] Red Hat Network Update Module. *Homepage*, September 2008. http://www.redhat.com/rhn/rhndetails/update.

[42] Relook - Digital Video Recoder. *Homepage*, September 2008. http://www.dgstation.co.kr.

[43] Report Linux kernel bugs via Bugzilla. *Homepage*, September 2008. http://bugzilla.kernel.org.

[44] RTAI - Linux Real-time Application Interface. *Homepage*, September 2008. http://www.rtai.org.

[45] RTLinux - Linux Real-time extension. *Homepage*, September 2008. http://www.rtlinux-gpl.org.

[46] Software Watchdog for Linux. *Homepage*, September 2008. http://sourceforge.net/projects/watchdog.

[47] Stress - Workload Generator. *Homepage*, September 2008. http://weather.ou.edu/~apw/projects/stress/.

[48] SUSE Linux Enterprise. *Homepage*, September 2008. http://www.novell.com/linux/.

[49] SUSE Linux Enterprise Real Time. *Homepage*, September 2008. http://www.novell.com/products/realtime/.

[50] SUSE Linux Enterprise Real Time overview. *Homepage*, September 2008. http://www.novell.com/products/realtime/overview.html.

[51] TiVo - Digital Video Recoder. *Homepage*, September 2008. http://www.tivo.com.

[52] TLDP - Software RAID HOWTO. *Homepage*, September 2008. http://tldp.org/HOWTO/Software-RAID-HOWTO-8.html.

[53] Ubuntu GNU/Linux. *Homepage*, September 2008. http://www.ubuntu.com.

[54] uCLinux. *Homepage*, September 2008. http://www.uclinux.org.

[55] User Mode Linux. *Homepage*, September 2008. http://user-mode-linux.sourceforge.net.

[56] The Wonderful World of Linux 2.6. *Homepage*, September 2008. http://kniggit.net/wwol26.html.