Preliminary assessment of Linux for safety related systems

Prepared by
CSE International Limited
for the Health and Safety Executive 2002

RESEARCH REPORT 011

Preliminary assessment of Linux for safety related systems

Eur Ing R H Pierce MSc CEng MBCS
CSE International Limited
Glanford House
Bellwin Drive
Flixborough
North Lincolnshire
DN15 8SN
United Kingdom

The Linux operating system is in widespread use, and there is now interest in using Linux for safety related systems. This report considers the availability and quality of evidence for the safety integrity of Linux. Three criteria are defined for the suitability of an operating system for use in safety related applications, namely that the operating system must be sufficiently well understood, that it must be suitable for the characteristics of the safety related application, and that it must be sufficiently reliable. Linux is then assessed against these criteria, and a framework for the hazard analysis of the interaction between applications and operating system is given. The overall conclusion of the study is that Linux would be, in broad terms, suitable for use in many safety related applications with SIL 1 and SIL 2 integrity requirements, and that certification to SIL 3 would be possible. However, it is not likely to be either suitable or certifiable for SIL 4 applications. An outline programme for the work necessary to certify Linux to SIL 3 is described.

This report and the work it describes were funded by the Health and Safety Executive (HSE). Its contents, including any opinions and/or conclusions expressed, are those of the author alone and do not necessarily reflect HSE policy.

HSE BOOKS

© Crown copyright 2002
Applications for reproduction should be made in writing to:
Copyright Unit, Her Majesty's Stationery Office,
St Clements House, 2-16 Colegate, Norwich NR3 1BQ

First published 2002

ISBN 0 7176 2538 9

All rights reserved. No part of this publication may be
reproduced, stored in a retrieval system, or transmitted
in any form or by any means (electronic, mechanical,
photocopying, recording or otherwise) without the prior
written permission of the copyright owner.

ACKNOWLEDGEMENTS

The author of this report gratefully acknowledges the contributions of his colleagues and Linux users Alastair Faulkner and Tim Rowe; and of Professor J A McDermid and Mark Nicholson of the Department of Computer Science, University of York.

FOREWORD BY HSE

A key element of a computer based system for protection or control is the operating system. There is a growing trend to base safety systems on general purpose commercial products such as Linux and Microsoft Windows rather than on operating systems developed specially for safety. Statutory regulators and other assessors of computer based safety systems need a practical procedure for assessing the safety integrity of a commercial operating system.

This report describes preliminary work in a collaboration between three UK agencies: the Defence Procurement Agency of the Ministry of Defence; the Civil Aviation Authority; and the Health and Safety Executive.

The goals of this collaboration are:

- To develop a description scheme which can be applied to a wide range of commercial operating systems, and which will permit an assessment of the key operating system attributes that are relevant to its use in a safety application.

- To develop a forecasting model to estimate the cost of achieving a specific level of confidence in the behaviour of an operating system.

- To assess in detail one or more operating systems of commercial importance in safety applications.

Preliminary work on this programme has focussed on two operating system of commercial importance which have been proposed for safety applications – Microsoft Windows XP, and Linux. For each a scoping study was made to investigate the practical aspects of the above goals. The findings of the two scoping studies will be evaluated to decide how to proceed to a full operating system assessment.

This Linux report should be read in conjunction with a companion report on Microsoft Windows XP: “Windows XP - Scoping Study for the Operating Systems - Integrity Evaluation of Windows”, produced by Praxis Critical Systems under contract to the Defence Procurement Agency, but which for administrative convenience is published by the Health and Safety Executive.

CONTENTS

EXECUTIVE SUMMARY

1 INTRODUCTION 1

2 CONTEXT AND SCOPE OF STUDY 3
Exclusion of the X Windows System 3

3 SAFETY AND INTEGRITY REQUIREMENTS FOR AN OPERATING SYSTEM 5
Reliability and safety integrity (criterion C3) 6

4 DESCRIPTION OF THE LINUX SYSTEM 7
Linux: a brief introduction 7
Linux facilities 8

5 PRELIMINARY ASSESSMENT OF LINUX AGAINST THE SAFETY CRITERIA 11
Well definedness of Linux behaviour (criterion C1) 11
Suitability of Linux for safety related applications (criterion C2) 13

6 FAILURE ANALYSIS OF LINUX FACILITIES 19
Functions and Related Calls for ARINC 653 20
Assessment of Linux Provision 23
Failure analysis strategy and example 23
Summary 24

7 LINUX CONFIGURATIONS AND APPLICATION EXAMPLES 27
Modular Kernel 27
Monolithic Kernel 28
Networked systems 28
Distributed system(s) 29
Emulated (replacement or upgraded) systems 30
Real time Linux 30
Platforms 31
Typical applications for which Linux may be useful 31

8 LINUX RELEASE AND CONFIGURATION MECHANISMS 33

9 STRUCTURE OF THE LINUX SYSTEM 35

10 SOURCES OF EVIDENCE FOR SAFETY INTEGRITY 37
Field service experience 37
Testing 39
Analysis 41

11 METHODOLOGY FOR LINUX CERTIFICATION 43
Effort estimates 44
Certification of other operating systems 44

12 CONCLUSIONS 47

13 RECOMMENDATIONS 49

REFERENCES 51

APPENDIX A Linux API functions 53

APPENDIX B Mapping Linux functions to the IMA service model 75

APPENDIX C Linux code metrics 81

GLOSSARY 86

EXECUTIVE SUMMARY

In recent years, the Linux operating system has been widely adopted by major organisations as the basis of their information technology infrastructure and product range. There are clear advantages in using Linux, for example a lower cost of acquisition than that for proprietary operating systems, wide availability on different platforms and avoidance of dependence on one supplier. Linux is also perceived as being reliable, and the availability of the Linux source text provides the ability to modify the operating system and to correct defects where necessary.

There is now interest in using Linux in safety related applications. This report has been commissioned by the UK Health and Safety Executive (HSE) under contract number 4383/R38.036 and is sponsored by HSE, the UK Ministry of Defence (MoD) and the Safety Regulation Group (SRG) of the UK Civil Aviation Authority.

The objective of the study as set out by HSE was to carry out a scoping study into the availability and quality of evidence to assess the safety and integrity of the Linux operating system.

The report sets out three basic criteria to decide whether an operating system is suitable for use in a safety related system, namely that the operating system must be sufficiently well understood, that it must be suitable for the characteristics of the safety related application, and that it must be sufficiently reliable.

Linux is assessed against the first two of these criteria and it is concluded that it would be suitable for use in some classes of safety related applications, provided that they do not have to meet very stringent timing requirements typical of "hard real time" systems. A framework for conducting a hazard analysis of the use of Linux in any given safety related system is introduced.

Sources of evidence that Linux is sufficiently reliable are considered, namely field service experience, testing and analysis. There is considerable field service experience available although much of this is of an anecdotal rather than a formal nature. Linux has been formally tested by a number of organisations and therefore some testing evidence is available. It is concluded by the study that existing test suites would form a good basis for a project to certify Linux for safety related applications, but that there is a need for a single specification, covering all aspects of behaviour including robustness, against which to test. The role of analysis in certifying Linux is considered, with a directed manual inspection process being recommended.

The overall conclusion of the study is that Linux would be, in broad terms, suitable for use in many safety related applications with SIL 1 and SIL 2 integrity requirements, and that it certification to SIL 3 might be possible. However, it is not likely to be either suitable or certifiable for SIL 4 applications.

An outline work programme is given for a possible Linux certification project with some approximate effort estimates.

1 INTRODUCTION

In recent years, the Linux operating system has been widely adopted by major organisations as the basis of their information technology infrastructure and product range. There are clear advantages in using Linux, for example a lower cost of acquisition than that of proprietary operating systems, wide availability on different platforms and avoidance of dependence on one supplier. Linux is also perceived as being reliable, and the availability of the Linux source text provides the ability to modify the operating system and to correct defects where necessary.

There is now interest in using Linux in safety related applications. This report has been commissioned by the UK Health and Safety Executive (HSE) under contract number 4383/R38.036 and is sponsored by HSE, the UK Ministry of Defence (MoD) and the Safety Regulation Group (SRG) of the UK Civil Aviation Authority.

The role of an operating system is to manage the hardware resources of the computer on behalf of application programs and users. An operating system forms a layer between the application programs and the hardware resources, and any failure in the operating system will almost inevitably lead to total failure of all applications running on the given hardware. The integrity of an operating system is thus crucial to the integrity of any safety related system in which that operating system is used.

The objective of the study as set out by HSE was to carry out a scoping study into the availability and quality of evidence to assess the safety and integrity of the Linux operating system. HSE and the other project sponsors are considering a possible project to certify Linux (or another operating system) for use in safety related systems. The information in this report is intended to support the sponsors in deciding whether such a project is necessary or feasible, and if so how it might be carried out.

Note that the term "qualification" is used in some industries, notably aerospace, to mean providing evidence that a given component does what it was specified to do, while "certification" is used to mean demonstrating that a system is safe. In this report the term "certification" is used with the former meaning, namely providing arguments and evidence that an operating system is fit for use in a safety related system, with suitable caveats and restrictions.

The structure of this report is as follows. The scope and context of the study are set out in section 2. Section 3 discusses the concept of safety and integrity requirements for an operating system and establishes three criteria which must be satisfied if an operating system is to be deemed suitable for a given safety related system. Section 4 gives a general description of Linux facilities for the benefit of readers who are not familiar with Linux or Unix, and section 5 then considers the general suitability of Linux for use in safety related work based upon the criteria established earlier. Section 6 provides a framework for failure analysis of Linux or indeed any other operating system, and uses this framework for a further assessment of the suitability of Linux for safety related applications. Section 7 describes Linux configurations, platforms and typical applications, while section 8 describes the Linux release and distribution mechanism. Section 9 provides some details on the internal structure and workings of Linux, to support other sections of the report. Section 10 discusses the currently available sources of evidence for the safety integrity of Linux. Section 11 considers how a possible project for Linux safety certification could be carried out. Finally, the overall conclusions of the report are given in section 12 and the recommendations in section 13.

Appendix A provides a list of the Application Programming Interface (API) functions provided by Linux.

Appendix B provides supporting information for the failure analysis scheme described in section 6.

Appendix C lists the results of some complexity measurements made on the Linux kernel.

It should be noted that this study is based on Linux kernel version 2.4 from an "out of the box" Linux distribution (section 8), which will be the most attractive option for many projects which do not have the knowledge or budget to modify Linux at the source code level. Special modifications of Linux are mentioned where relevant but have not been considered in detail.

It should also be noted that Linux distributions typically provide large numbers of application programs and other ancillary software. This study is only concerned with the Linux kernel and other operating system components, together with the necessary libraries to allow access to the operating system, and not with user-level applications.

2 CONTEXT AND SCOPE OF STUDY

The work described in this report draws upon an earlier study of the requirements for operating systems in safety related applications carried out by CSE on behalf of SRG [Ref 1]. It also uses an evidence-based assessment framework for safety related systems which is given in the SRG document CAP 670 section SW01 [Ref 2].

The study was carried out by means of literature and World Wide Web searches, study of textbooks on Linux facilities and structure, personal and company experience of using Linux, manual examination of samples of the Linux kernel code, and some static analysis of the Linux code.

The term "Linux" strictly only applies to the Linux kernel, but is used in this report to refer to the operating system as a whole including device drivers and filestore support, together with the necessary API libraries.

2.1 EXCLUSION OF THE X WINDOWS SYSTEM

The X Windows system is commonly used with Linux to provide the user interface on high resolution graphical workstations, which include most PCs in common use. The X Windows system as been excluded from detailed consideration in this study for several reasons:

i) X Windows has been extensively used with Unix operating systems, in addition to Linux, for many years, and has accumulated an even greater amount of field service experience than Linux itself, including use in safety related systems;

ii) X Windows runs as an application, which does not require the operating system to be modified and can thus be assessed separately from Linux;

iii) it is not necessary to use X Windows to provide the user interface on a graphical workstation or PC, the "curses" package supplied with Linux, or other similar packages, can be used instead to provide a simpler user interface; and

iv) a safety related application can protect itself against failures in X Windows, whereas it cannot protect itself from failures in Linux proper (without using hardware assistance).

It can be argued that X Windows and Motif (a layer of software on top of X which provides a particular window manager and "look and feel" to the GUI) are mature items of software which are currently used in many safety related applications with SIL 1 and SIL 2 integrity requirements. For example, one SIL 2 system with which CSE is familiar in detail has accumulated over 90,000 hours of operating experience without any failure attributable to X Windows or Motif. A defect reporting system provides confidence that any failure would have been recorded and analysed.

CSE therefore concluded that spending effort on further analysis of X Windows in this study would not be justified.

Note that the remarks above about the reliability of X apply to the original implementations of X Windows and Motif. Any implementations which provide the same facilities and look and feel but which do not use the same source code (with the exception of hardware dependent components) will not have this record of successful use.

3 SAFETY AND INTEGRITY REQUIREMENTS FOR AN OPERATING SYSTEM

There are a number of practical problems with using an operating system in a safety related environment [Ref 3]. The first is that there is no accepted way of specifying or describing the complete safety related behaviour of an individual software component. For this reason, defensive mechanisms need to be employed against relevant failures. As noted in the SRG report [Ref 1], there is a major difference between an operating system and any other kind of pre-existing software component in that the operating system provides a layer between the application and the hardware. Failure of the services provided by the operating system to the application program will therefore inevitably result in application software failure (note of course that the operating system could have internal fault containment and recovery behaviour) and the application software cannot therefore provide such defensive mechanisms. In some cases, hardware support may be required to implement suitable defensive mechanisms against operating system failure, and this is discussed further in section 6.

It is argued, therefore, that for an operating system (or indeed any pre-existing software) to be suitable for use in safety related system, it must satisfy the following criteria:

C1 the behaviour of the operating system must be known with sufficient exactness, in all relevant domains of behaviour, to provide adequate confidence that hazardous behaviour of the safety related application does not arise because of a mismatch between the belief of the application designer and the true behaviour of the operating system;

C2 the behaviour of the operating system must be appropriate for the characteristics of the safety related application, in all relevant domains of behaviour; and

C3 the operating system must be sufficiently reliable to allow the safety integrity requirements of the application to be met (when taken together with other system features). In other words, the likelihood of failures of the operating system features and functions used by the application must be sufficiently low.

While system testing would in general reveal many failures caused by a mismatch between designer understanding and the true behaviour of the operating system, it cannot guarantee to eliminate all such mismatches which could cause the safety related system to fail in operation. The higher the integrity requirements of the application, the more important C1 becomes.

A corollary of criteria C1 and C2 above is that, where there are defects in the operating system or limitations that may make it unsuitable for certain safety related applications, these must be known to the application designer so that they can be taken into account in the choice of the operating system and the design of the application software.

A safety case for the use of Linux in a given safety related application must address these three criteria, although the structure of such a safety case is beyond the scope of this study. Criteria C1 and C3 can be addressed by “certification” of the operating system, which is the subject of this report. Criterion C2 is the responsibility of the application designer, who must show that the behaviour of the operating system is appropriate for the application in question. Section 6 provides more detail on how this can be achieved.

3.1 RELIABILITY AND SAFETY INTEGRITY (CRITERION C3)

In IEC 61508 [Ref 4] and in some other standards, a number called the Safety Integrity Level (SIL) is defined for the functions of a safety related system on the basis of risk reduction requirements. A mapping is provided between the tolerable hazard occurrence rate (for continuous or high demand systems) and the target SIL. The safety related functions are allocated in the system design to system elements including hardware and software, and corresponding reliability requirements are apportioned to those same system elements. Depending on the system design, the failure rate target (and corresponding SIL) of a component may be higher or lower than the SIL of the overall safety function (this depends whether components are in a series or a parallel, redundant configuration).

For a software component (either a complete software system running on one computer, or one element in a partitioned software system), the SIL of the software is determined by its apportioned failure rate using a mapping from target ranges of failure rate to SIL. In some standards such as DO-178B [Ref 5] the equivalent of the SIL is derived from failure consequence rather than risk, but this makes little difference to the overall principle.

Clearly any operating system used for an application of a given SIL must have a SIL which is at least as high as that of the application, and for the sake of confidence one higher would be desirable, from the argument given at the start of section 3 above.

The shorthand term “A SIL N operating system” therefore means “an operating system whose failure rate is demonstrated by adequate evidence from appropriate sources to be at least as good as that required to support SIL N functions, having regard to system architectural and failure defence mechanisms”.

Since this definition is cumbersome in use, the shorthand form “a SIL N operating system” may be used.

4 DESCRIPTION OF THE LINUX SYSTEM

Linux is a general purpose operating system which is intended to support both program development and the execution of application software in a production environment. This brief description of Linux background and facilities is intended as a guide to those who are not familiar with the Unix model of an operating system.

4.1 LINUX: A BRIEF INTRODUCTION

The Unix system was first described by Ken Thompson and Dennis Ritchie in 1974. Unix development began in 1969 at Bell Labs. After 30 years the Unix system and the concepts which it embodies, are now present and extended in the Linux operating system. Unix has been widely used in Universities throughout the world where major modifications have lead to the divergence of the system functionality through the evolution of the design. The University of California at Berkeley, USA was largely responsible for the introduction of networking capability through the addition of “sockets”. Although Bell Labs (AT&T) retained the licensing rights to Unix and co-ordinated major revision and design changes, by the time these controls were instituted Unix had escaped into the academic community and proved difficult to control.

Unix possesses a simplicity and clarity which has both facilitated a large developer community and allowed those developers to enhance it in their own way. The Linux kernel was originally designed by Linus Torvalds whilst he was attending the University of Helsinki and later developed through the collaboration of many volunteers worldwide (through the use of the Internet). Most versions of Linux cannot technically be referred to as a version of Unix as they have not been submitted for test and subsequent licensing.

The great advantage that the implementation of Unix has retained from its early days is that a majority of the kernel is written in the relatively high level language C, with only the hardware specific interfaces written in the assembler (but see section 10.3 for notes on safe subsets of C). This early design decision has provided a platform for the definition of a virtual machine interface, which has facilitated the porting of Unix from its original PDP-7 and PDP-11 platforms to many other hardware architectures.

As noted earlier, the term Linux as used in this report refers only to the operating system. In common use the term Linux often refers to a distribution (see section 8), which contains the kernel, utilities, and in many cases a substantial number of applications, documentation and source code. Linux offers all the common programming interfaces of standard Unix systems. The Linux utilities are provided through the GNU project run by the Free Software Foundation. These utilities are, in the main, text based and include development tools such as compilers and libraries. The third major contribution to a Linux distribution came from the X Consortium, which developed the X Windows system, and the Xfree86 project which ported the X Windows system to standard PC hardware (graphics cards and devices).

The Linux operating system has now been in existence for over 10 years. Many Linux distributions are available from simple, feature-restricted, embedded systems to large distributed computing environments. For the office environment an off-the-shelf Linux distribution for installation would present the user with a graphical user interface similar to most modern office computing environments.

The Linux kernel design, management and co-ordination are still undertaken by Linus Torvalds and an inner circle of developers who maintain the design intent. The kernel source code is under the control of the CVS source code management system.

4.2 LINUX FACILITIES

A summary of the Linux application programming interface (API) functions is given in Appendix A. The API provides an interface at the programming language level (C functions) to the system calls which invoke the Linux kernel. Other behavioural characteristics of a Linux system are configured or read by standard text files and by means of a special "proc" file system.

Processes Application programs are run as processes under Linux. Each process has its own virtual memory and processes are scheduled for execution on a time-sliced, priority based scheduling system which always provides some processor time to any process. Processes are constructed in parent/child hierarchy; if one process starts another (by means of the well-known "fork" system call) then the new process is the child of the original process (and the parent process is notified, by means of a signal, when the child terminates). The first process in the system is started by Linux and this creates other processes by means of an initialisation script. One process started in this way would typically be for user interaction via the keyboard and mouse.

Linux now supports symmetrical multiprocessor configurations in which processes can be genuinely executing in parallel on multiple hardware processors.

Threads A recent addition to Linux (and to Unix systems in general) is the idea of a lightweight process or "thread". One process can be executing many threads, with differing priorities for each thread. All threads share the same virtual memory space and code. Internal thread synchronisation facilities are provided. Being relatively new, threads are relatively little used by comparison with normal processes. An Ada compiler will typically use threads to implement Ada tasks.

Users Unix was originally developed as an interactive timesharing system for multiple users. Linux therefore supports the concept of "users" with user names (UIDs) and passwords for the purpose of protecting the resources (generally the files) of one user from access by other users. Users can be collected into groups with a group name (GID). Users can grant access to their files to other users in a user group or to any user. There is a superuser concept (user name "root") where the superuser can access or change any system resource. Although users are frequently individuals who use the computer, the concept can be used in an abstract manner, for example to allow only programs of a particular class to access certain resources (an example is given in section 5). Linux can therefore support some forms of partitioning.

Input/output services A central concept in Unix and Linux is the filestore: a hierarchially named collection of files with defined user and group access permissions. In Linux a "Virtual filestore" is implemented via the kernel, so that a variety of different filestore implementations can co-exist on the same system. Files are logically named by paths starting at the root of the filestore "/ ", for example "/etc/fstab". Hardware devices are, as in Unix, addressed as special classes of file (/dev/…). Files are treated as linearly addressable, hiding the specific organisation of discs, tapes and other peripheral devices. An error is returned if a file operation is not applicable to the file being accessed. Device or file I/O can be either synchronous (the process blocks until the I/O operation is complete) or asynchronous (the process continues while the operation is in progress). Various means are provided to determine whether an asynchronous operation has completed. The most useful is the "select" operation which enables the process to test whether one or more given I/O operations are complete. The select call can also optionally cause the process to block until either an I/O operation is complete, or a signal of any kind is received.

Input and output services also importantly include network access by means of the socket concept. A socket is the means by which a user application sends and receives messages to other computers on a local or wide area network. Sockets provide the interface to Internet protocols including TCP/IP, UDP/IP and raw IP. Both sockets and files are accessed via a "file descriptor" which identifies the file, device or socket being accessed.

Inter-process communication (IPC) IPC is provided by a variety of means:

i) signals (software interrupts to a process);

ii) pipes (destructive read files), both named and anonymous;

iii) semaphores (for processes);

iv) shared memory;

v) local sockets (a socket interface to process on the same processor);

vi) mutexes (for mutual exclusion of threads from critical areas); and

vii) condition variables (to allow threads to suspend themselves until the variable has a given value).

Note: mutexes and condition variables provide semantics very similar to the protected object concept in Ada 95 [Ref 6].

Timer services A process or thread may define several interval timers (expiry of the timer is indicated by a signal to the process). There is also a facility for processes and threads to suspend themselves for a specific time period, and a facility to get the time of day.

Command line interpreter (shell) A command line interpreter or shell is available for basic user interaction. Although in principle the shell is simply an application run on top of the Linux API, it has a special role in the Linux system, for example on system start-up. A process can be made to run a shell script (as opposed to an executable binary program), and this is a facility which is frequently used in Unix and Linux systems. The standard shell provided with Linux is "bash" which provides the functions and syntax of the Bourne shell in Unix.

5 PRELIMINARY ASSESSMENT OF LINUX AGAINST THE SAFETY CRITERIA

This section provides a preliminary assessment of Linux against criteria C1 and C2. Criterion C3 concerning safety integrity is discussed at greater length in section 10.

5.1 WELL DEFINEDNESS OF LINUX BEHAVIOUR (CRITERION C1)

The domains of behaviour for a software system in general are identified in SW01 [Ref 2] as follows:

i) Functionality;

ii) Timing and Performance;

iii) Capacity;

iv) Failure Behaviour (of the system itself/connected systems/user programs);

v) Overload Tolerance;

vi) Reliability (safety integrity, as in criterion C3 above); and

vii) Accuracy (of numeric computation).

There is, as far as this study has been able to determine, no single definitive work which describes the services provided by Linux under the headings of all the above SW01 attributes. Unix implementations normally provide machine readable "manual pages" which define various aspects of operating system calls, shell commands and standard utility functions or programs. The manual pages for typical Unix systems often lack the precision that would be thought necessary for safety related systems, as the author of this report knows to his cost. For Linux, manual pages are being replaced by "info" pages.

It should be noted that such manual pages generally only describe the parameters and immediate effect of calls to the operating system API rather than specifying the behaviour of the operating system in wider terms. Operating systems are very "stateful" machines in the sense that the behaviour of the operating system and its responses to system calls is strongly influenced by the previous history of system calls made by the set of executing application processes. This is one reason why operating system behaviour is difficult to define in detail. In addition, some aspects of operating system behaviour may be controlled by means other than API calls, for example by configuration data files.

Linux claims to conform to the POSIX definition [Ref 7]. POSIX is a basic standard for Unix-like operating systems which specifies the syntax and semantics of system calls which a conforming operating system must provide to the user application program, and some other aspects of operating system behaviour. 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).

There are many textbooks describing the behaviour and administration of Unix and Linux systems and some 30 years of running Unix in various varieties; its general behaviour and interfaces are thus well understood by many developers. There are also books which describe Linux internals (for example [Ref 8, Ref 9]) which can contribute to an understanding of aspects of behaviour which may not be clear from the POSIX or other definitions.

A useful source of documentation for Linux is provided through the Linux Documentation Project (LDP) [Ref 10]. As well as providing references to textbooks, volunteers support Linux by providing a range of documents known as "howtos", "mini-howtos" and FAQs. Howtos deal with specific administration or installation issues such as the configuration of a modem to provide access to the Internet by phone. Mini-howtos are abbreviated howtos and summarise the larger documents for those who do not wish to be overwhelmed with detail. FAQs are "Frequently Asked Questions" and record the most popular questions asked, originally on bulletin boards or mailing lists.

Finally, with an open source system an appeal can ultimately be made to the source code to discover some aspects of behaviour which are obscure, although this would be a difficult task for those not expert in operating system construction (but Linux textbooks and the LDP do provide assistance in this respect).

The Linux Professional Institute [Ref 11] is concerned with the certification of Linux staff competence through an examination process. Certification is further supported by the Linux Training Materials Project [Ref 12] which aims to provide a single source of training materials. Competence of project engineers to understand and use Linux could be one part of a safety argument for the use of Linux in a safety related application.

Differences in behaviour between different distributions of Linux can currently arise because of differences in the API application libraries supplied by different Linux distributors. Some of these differences are minor but have an effect on portability of applications between different platforms. The Linux Standard Base project (LSB) [Ref 13] is currently defining a standard of behaviour for Linux distributions. If such a specification is agreed, it should be used for any certification project.

Reported errors in Linux, fixes and "work arounds" are recorded and presented for inspection on web sites such as [Ref 14]. Clearly, any safety related project would have to take a view as to whether it was better to avoid known errors or adopt a new version of Linux in which those errors were corrected but which might have unknown new errors.

Timing and performance are generally related to a particular computer configuration and are not therefore describable in general terms.

Some aspects of failure behaviour are specified, in particular the responses made by Linux to incorrect system calls. Other "non-functional" attributes of Linux behaviour are not in general defined.

It can be concluded, therefore, that although the general behaviour of Linux is reasonably well described by a number of information sources, there is no single definition against which it can be validated, unless the Linux test or documentation projects provide such a definition in future.

Note also that specific device drivers will have specific behaviours in response to general Linux file manipulation commands, and these behaviours would need to be defined for a safety related system.

A Linux certification project should therefore provide a single definition (which could probably be based on the POSIX standard with extensions) to clarify the behaviour of Linux under the SW01 attribute headings as applicable. Although a mathematically formal definition of the Linux kernel behaviour is probably feasible, it is unclear what value would be gained from a fully formal definition. However, the definition should be as formal as is useful and reasonably practicable. The definition would both provide a basis for testing, and to form a baseline for users of the certified Linux. It should also clearly state what is not known, for example device dependent behaviour, and provide the developer of a safety related application a framework in which such behaviour can be defined.

5.2 SUITABILITY OF LINUX FOR SAFETY RELATED APPLICATIONS (CRITERION C2)

This section considers the suitability of Linux for safety related applications, given that an adequate definition of its behaviour is available. Any individual project using Linux for a safety related application should however carry out a detailed study into its suitability. Section 6 addresses the question of how a particular project can carry out such an analysis.

5.2.1 Functionality

Section 3 above provides a general description of the facilities offered by Linux.

This section discusses some aspects of functionality which are of particular relevance for safety related systems.

5.2.1.1 *Partitioning*

The ability to provide mechanisms to manage resources safely when a number of programs are co-operating is crucial for an operating system used for safety related applications (unless the entire application is written as one process). This is known as partitioning. Rushby has written a lengthy paper [Ref 16] discussing partitioning based on the civil Integrated Modular Avionics (IMA) model. Rushby describes partitioning as a mechanism to prevent fault propagation, but notes that this is only a protection mechanism from new hazards created by the sharing of resources. Other hazards such as the incorrect calculation of an output value by an application are not covered. He divides the partitioning problem into two aspects:

i) **Spatial Partitioning** This prevents a partition altering another partition's data or software, and also prevents command of another partition's associated output devices; and

ii) **Temporal Partitioning** This ensures a partition receives services, such as access to the processor or timely access to a physical device, which are unaffected by other software.

Partitioning should ideally provide fault containment equivalent to a system in which each partition was running with its own dedicated hardware and resources. The behaviour and performance of software in one partition must be unaffected by the software in other partitions.

In practice, on a uniprocessor system, the CPU usage of one partition must inevitably affect the time available to other partitions but need not affect other resources.

Linux processes provide a useful spatial partitioning mechanism, by providing a separate virtual address space and resource protection mechanism for each user process. Clearly this depends on the presence of a suitable memory management unit, but this is available on all the computers on which Linux is currently implemented with the exception of some embedded systems. This facility means that one process should not in principle be able to modify the program or data memory of another process. The Linux shared memory facilities need co-operation between the processes using it.

Device drivers could corrupt memory when writing data from kernel space to user space. The integrity of these needs to be assured, since they are an integral part of the operating system.

User processes can access real memory with the special file “/dev/mem”; however, this needs root privilege.

Since one process cannot rely upon another not to interfere with its memory in this way, all processes need to be analysed (and in general only a supervisory process should be run as “root”). In this sense Linux, in common with Unix in general, is often known as a “trust based” system.

Spatial partitioning can be reinforced by the UID and GID concepts (section 3). For example, the “Postgres” database requires that it and its associated data management applications and utilities be allocated a specific GID and UID to reduce the possibility of access and modification by other users. This requirement is enforced by each utility testing for the UID of the Postgres user, at initialisation, and terminating with the appropriate error message if the application is run as any other user (including the superuser).

Temporal partitioning is, to some extent, provided by the time slicing behaviour of Linux. This will not guarantee to give processes access to the processing resource at a defined time but will prevent any process from being completely starved of processing resource. The process priority mechanism can be used to ensure that higher priority processes obtain more CPU time than lower priority processes. Temporal partitioning is therefore rather weak in Linux.

5.2.1.2 *Autonomous behaviour*

Operating systems are capable of displaying autonomous behaviour (in other words, behaviour which is not directly commanded by the applications which are currently running). Such behaviour could include performing housekeeping functions at a particular time of day or when some internal limit is reached. Provided that the autonomous behaviour does not fail, the impact of such behaviour will generally be to increase the response time for services requested by the application, perhaps by a large amount, thus causing “jitter” in response times to input demands, or variations in throughput.

As will be noted later, Linux does not exhibit extensive autonomous behaviour and is quite suitable for soft real time systems. One aspect of autonomous behaviour is the standard filestore mechanism of buffering files in memory to avoid writing back to disc on every write to the file. A continually running buffer flush process is used to write buffered pages back to disc at intervals typically of a few seconds. This behaviour can be disabled by changing the initialisation files but at the expense of considerable reduction in disc I/O performance.

The standard “cron” daemon which is used to carry out functions to a predefined schedule is under user control by means of a configuration file and need not be used if not required.

5.2.1.3 *Omission of unused functions*

In any safety related application it is desirable in principle to remove unused code so that there is no chance of it being invoked in unforeseen circumstances and perhaps causing unexpected behaviour. Since Linux is supplied in source form, it is possible in principle at least to ensure that only relevant device drivers and filesystems are built into the kernel. The “monolithic kernel” variant of Linux (section 7) is the most useful in this respect.

5.2.2 Performance and timing

Applications can be broadly classified as Hard Real Time (HRT), Soft Real Time or non real-time.

HRT applications are those where the system is required to meet absolute and very short timing deadlines (of the order of a few milliseconds) and where failure to meet a deadline would represent a system failure (and in a safety related system, a system hazard). In a Soft Real Time system, a certain latitude is allowed in response times, although typically some maximum response time of the order of seconds might be required (see also under section 7). Non real-time applications are those which have no definite deadlines, and in safety related systems would generally be confined to off line data preparation tools.

The timer resolution on PC-based Linux systems is 10ms, which is not adequate for many HRT applications. This can be reduced by modifying the kernel, provided that the hardware timer resolution is suitable, but more frequent timer interrupts will reduce overall system performance.

What cannot be specified by a standard is the actual performance that a given Linux implementation will provide, since this will depend on processor speed, bus speed and other factors. For HRT operation, it is regarded as essential to provide predictable operating system overhead for kernel operations such as process switching, and for all system calls. The Ada 95 language standard specifies one such set of kernel timing requirements [Ref 6 Annex H]. Obviously disc I/O responses will depend on the current disc file size and organisation, and for this reason HRT systems typically do not use, or use very limited, disc I/O.

Another source of timing and performance variation can arise from virtual memory and demand paging. If the set of application process code is larger than the amount of RAM on the computer available (having regard for the operating system space and I/O buffers), pages may be swapped out and swapped in again later when accessed. This can cause considerable response jitter in the process whose code is subject to swapping. Programs can be marked as not to be swapped, and on modern computers the amount of RAM available is generally so large that demand paging is not likely to be encountered. Demand paging is not likely therefore to be a particular source of timing problems but should be borne in mind as a potential source of response time variation. HRT systems in particular should avoid any demand paging.

I/O operations consume buffer space, and the kernel will invoke memory housekeeping functions (such as searching for a free buffer or merging freed buffers to make a larger buffer). This can affect the time taken to respond to I/O operations and can also create response time jitter for other processes.

Dependable scheduling policies (such as deadline monotonic and rate monotonic) which will guarantee the meeting of timing deadlines depend on invariant process or thread priorities and on guaranteed operating system response times [Ref 17] and the standard Linux time sharing policy for processes does not meet these requirements.

Linux was not intended to support HRT behaviour and there is some evidence, apart from the above considerations, that it is not intrinsically suitable for such work [Ref 18, Ref 19] because a large amount of the operating system code is non-preemptible (where interrupts are disabled).

The conclusion reached is that Linux is not suitable for HRT work with very short deadlines, but would be suitable for Soft Real Time providing that the processor load is not excessive (15% loading under normal operations is a rule of thumb used by some system designers) and programs are not "too large" reliably to avoid demand paging.

Part of any certification project could be to obtain performance benchmark figures for the specific system calls and kernel operations under a mixture of loading conditions.

Most if not all SIL 4 applications have HRT behaviour defined (and generally do not run under any form of operating system or pre-existing kernel) and it is questionable whether Linux would be suitable for such applications, even if its reliability could be adequately demonstrated.

5.2.3 Capacity

Various resources are allocated within the kernel, for example the process table and the file descriptor table. On modern PC hardware, limits on internal Linux table sizes are large and can be changed by a knowledgeable user (by means of a kernel rebuild and, in the most recent versions of the kernel, dynamically). Capacity limitations are unlikely therefore to present any problems in safety related Linux applications.

5.2.4 Robustness

There are three aspects of robustness to be considered: robustness of the operating system to unexpected behaviour of user processes, robustness in the face of failures in peripheral devices, and robustness to failure of the hardware processing platform.

System calls generally check their parameters, and the kernel is unlikely to crash due to illegally-formed systems calls, although it may not provide an error indication in all cases. Some information is given in [Ref 20] on the robustness of Linux to system calls with unexpected parameters.

The kernel is protected by the virtual memory management system and kernel space is therefore not accessible to user processes.

Exceptions arising in user processes are trapped by the kernel and reported back to the process by means of a signal.

If the kernel itself raises an exception (such as accessing real memory outside limits, or arithmetic operation error) it generally does not recover and all application activity ceases. The kernel will attempt to output an error message. The kernel is not self-restarting (exceptions to this rule may appear on specific hardware types), and where rapid recovery is required the designer should therefore employ system measures such as watchdog timers.

The method of dealing with failures in connected peripheral devices depends on the device driver. In many cases a complete failure may be indicated by a timeout condition on an attempt to access the device, in which case an error message may be returned to the user program. Other, more subtle disc errors can cause Linux to crash. The study has not determined the detailed behaviour of Linux in these areas and this would be a useful aim of a certification project.

It is expected that failures in the underlying hardware platform (memory or CPU for example) could not be tolerated by Linux but this aspect of robustness has not been examined by the study. Obviously any hardware platform for use in a safety related system must be sufficiently reliable to meet the SIL requirements of the system.

5.2.5 Overload tolerance

If a large number of processes are running, this will result in longer times between the execution of each process but as noted above there is no hard limit, and no process will be completely starved of processing resource.

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.

5.2.6 Accuracy

This domain of behaviour has no meaning in the context of an operating system.

6 FAILURE ANALYSIS OF LINUX FACILITIES

In the SRG operating system requirements, as yet unpublished, there is a requirement to carry out a hazard analysis of the interaction between the application and the operating system, to provide assurance that use of an operating system does not present any new hazards at the application level.

The headings, under which the SRG requirements are grouped, are as follows:

i) Executive and Scheduling facilities;

ii) Resource Management;

iii) Internal Communications;

iv) External Communications;

v) Liveness;

vi) Partitioning;

vii) Real-time;

viii) Security;

ix) User interface;

x) Robustness; and

xi) Installation.

In the aerospace industry the ARINC 653 standard [Ref 21] puts forward an API for the operating system to application layer for safety related avionics systems, especially those which support the Integrated Modular Avionics (IMA) concept. It provides a good basis for assessment of the quality of an operating system for such systems. Work at the University of York has shown that failure analysis of this API can be undertaken. However, failure analysis of the system calls implied by the 653 API has shown that considering the failure of each call in isolation does not produce results in a form that is useful for assessing the failure characteristics of the overall system (consisting of a number of co-operating processes or partitions).

What is required is a functional failure analysis (FFA) of the operating system, the results of which can be plugged into the analysis of the failure characteristics of each application that relies on the functions of the operating system. For instance, a failure of the operating system may appear as a base event in the fault tree for a particular failure mode of an application. Thus it is necessary to determine a set of functions that the operating system must provide if an application is to provide the intended functionality, and then to undertake a failure analysis of these functions.

For the ARINC 653 API six generalised "functions" were determined for safety related applications. These six generalised functions can also be used a basis for undertaking a failure analysis of Linux as part of a study into the suitability of Linux for any particular set of applications. The functions can also be used to analyse other operating systems. Note that the approach recommended here also implies that the results of this analysis must be available to each application developer and used as appropriate in their safety analyses.

The six functions that need to be supported by an operating system for safety related systems are:

i) **provision of secure and timely data flow** to and from applications and I/O devices;

ii) **controlled access to processing facilities**. The access of applications to the underlying hardware processing resources must be managed so that, for example, any deadlines can be met;

iii) **provision of secure data storage and memory management**. The aim here is to secure memory storage from corruption or interference by other applications or the actions the operating system takes on their behalf;

iv) **provision of consistent execution state**. This concerns the consistency of data and is mostly concerned with the state of the system after initialisation;

v) **provision of health monitoring and failure management** covers partial and controlled failures of the system (operating system, application, hardware); and

vi) **general provision of computing resources**. This covers provision of any of the services of the O/S. A failure of this function would imply an uncontrolled failure of the O/S.

This is a slightly higher level classification than that given by the SRG operating system study.

At the higher end of the integrity level spectrum all of the six functions listed above will be required. In lower integrity systems, and for some classes of application, it may be possible to "water down" some of the requirements implied by these six functions. In this study, the system calls on the Linux API were considered and an attempt made to map them to the six functions listed above. Any extra calls were put into an "other" category. The study then investigated how well the system calls in Linux appear to support these functions.

6.1 FUNCTIONS AND RELATED CALLS FOR ARINC 653

System calls are used to provide the functions needed by the applications. Analysing the calls for the ARINC 653 generic API led researchers to group together a number of system calls in the form of an operating system services classification. These services can be used partly to provide more than one function. In Figure 1 the set of six functions are mapped to services chosen for the ARINC 653 study [Ref 22]. For example, Figure 1 shows that in order to provide controlled access to processing, the services of *scheduling*, *timing watchdog*, *initialisation*, *processing*, *configuration management* and *close-down* are all required.

This classification of services is fairly similar to the taxonomy of operating system services included in the SRG study [Ref 1] and listed at the head of section 6 above, but includes specific items such as timing watchdogs. The ARINC model, with its two levels of classification (functions and services) may be regarded as superseding the model in the SRG study. However, the SRG study classification could be used by a project as a checklist to ensure that no aspects of operating system behaviour have been omitted from consideration.

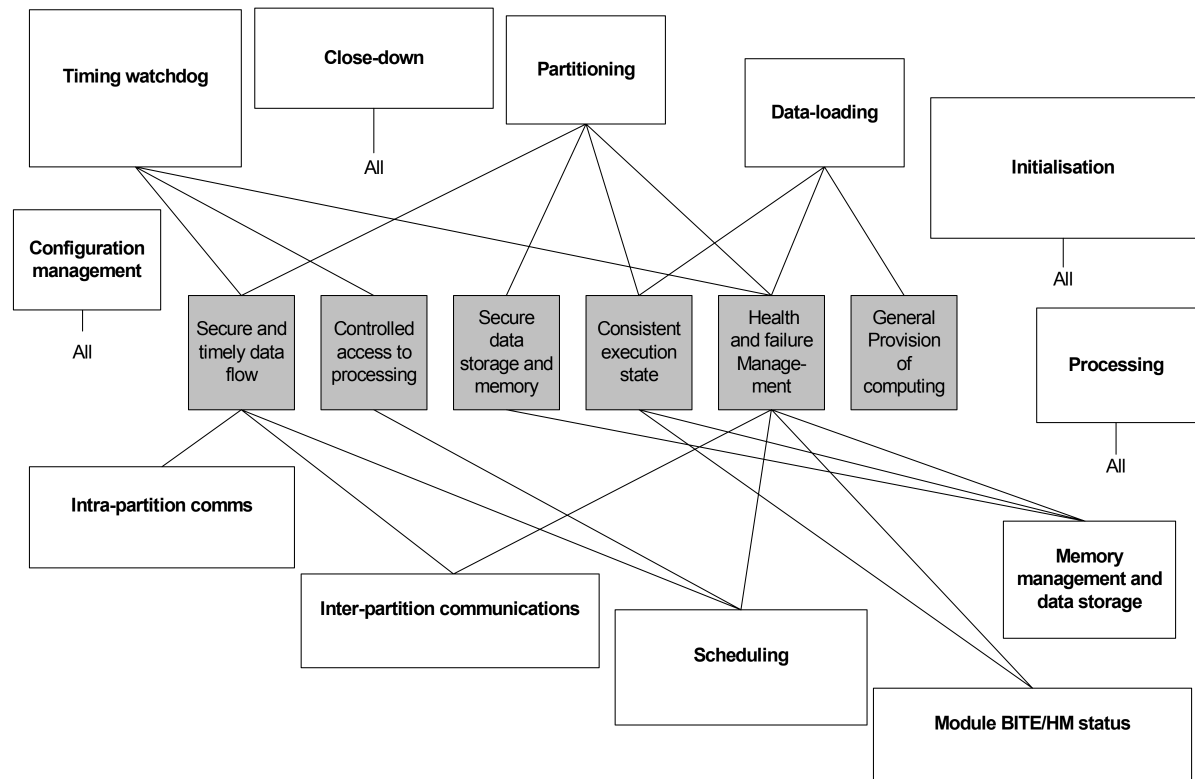


Figure 1: Functions and related calls For ARINC 653

This view of operating system services can be used for the analysis of the acceptability of Linux, or any other operating system, for use in safety related applications. Clearly, if the services are not provided, or their provision is weak, the functionality required of the operating system by an application may not be available. This view complements the analysis given in section 5 which is based upon the SW01 attributes.

A safety study of a system for which a pre-existing operating system such as Linux is proposed should assess the services provided and how this affects the ability of the operating system to provide appropriate functionality to support a given application.

The services in the ARINC 653 model are as follows:

i) **Data loading**: allows data to be installed on the system and to be loaded for execution (and for restart after failure);

ii) **Initialisation**: provides complete system, process, thread, semaphore etc initialisation. Initialisation may be on start-up or error recovery;

iii) **Timing watchdog**: provides for processes to have real-time deadlines and can detect failures to meet these deadlines;

iv) **Partitioning**: if data, spatial or timing partitioning of processes is to be employed (to allow multiple integrity level processes to be managed by the O/S for instance) then services to allocate resources to these partitions and police their usage is required;

v) **Intra-partition communication**: if one or more processes / threads reside in the same partition then provision for passing data will be required. This could be via shared memory for instance. More than one method may be supported;

vi) **Inter-partition communication**: allows data to be exchanged between a number of partitions and between partitions and external I/O devices. Note that services 5 and 6 support inter process / thread communication;

vii) **Scheduling**: service allows processes / threads to demand processing time based on their priority;

viii) **Processing**: provision of data and instruction processing from the CPU;

ix) **BITE and Health monitoring**: data logging, error detection and recovery activities;

x) **Close-down**: shut down of processes, threads, partitions, hardware devices, mutexes, semaphores, files and other resources, and the complete system;

xi) **Memory management and data storage**: preserve data, ensure consistency and coherency of data. Note the overlap with the partitioning, initialisation and close-down services; and

xii) **Configuration management**: arrangement of processes and operating system support on the hardware architecture. Also comprises the provision of information such as the available communication ports, I/O devices attached etc. Any support for system reconfiguration will be placed in this service category.

It may be that a particular system call contributes to a number of services (these could take the form of wrapper functions or language specific functions in Linux). A failure of this system call will then constitute a common cause failure mode for a number of services. Similarly one service may contribute to a number of functions, as shown in Figure 1. This must be taken into account in the failure analysis of the functions.

Appendix B provides a mapping of Linux API functions to the services listed above, showing the degree to which the twelve operating system services, and by implication the six higher level operating system functions, are supported by Linux.

6.2 ASSESSMENT OF LINUX PROVISION

It is reasonable to assume for the purposes of this study that the functions and services defined above are required, in general, of any operating system for safety related systems. It should be noted, however, that the IMA concept has particularly stringent requirements for partitioning since the idea is to support applications from different suppliers in a single computer. The IMA concept also requires very precise timing support to allow each application partition timely access to a time division multiplexed data bus. Finally, built-in test and failure management are especially important for airborne applications.

The analysis in Appendix B indicates that two services are not well supported by Linux, as follows:

i) provision of secure and timely data flow – as already shown in section 5, the timing model inside Linux is weak. 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. Although one process can use the Linux timer facilities to determine if another process has responded to it within a given time period, this is a less powerful feature than a true watchdog timer; and

ii) health monitoring (including BIT) is poorly supported within Linux. This is likely to be a significant omission as it provides input to two important functions for safety critical systems. The mechanisms for failure management are also not clear from the Linux API, and in general device failure management is also weakly supported, as discussed in section 5. However, disc health monitoring which is provided by some drives using the SMART technology is supported by a Linux application [Ref 23].

This brief examination of the required functions, services and system calls using the ARINC 653 model indicates clearly the potential weaknesses of the Linux operating system for some classes of safety related application. The systematic failure analysis framework given below can be used to draw these weaknesses out and put forward requirements to address these deficiencies in a particular system. The severity of the weaknesses will vary with the application context, for example unmanned systems will in general have more stringent requirements for BIT, health monitoring and watchdog timer services than systems where human supervision is available.

6.3 FAILURE ANALYSIS STRATEGY AND EXAMPLE

Each of the six generalised functions identified above needs to be analysed to identify possible failure modes, to consider their causes and to identify derived requirements to deal with the failures. Since the basis of this study is to assume that Linux will be acquired off the shelf, the derived requirements must be placed either on the hardware infrastructure, the applications software, or specialised software such as new or modified device drivers. In any real safety related application, a detailed study would be required. This section indicates the approach that should be taken and to give an example fragment of an analysis and the derived requirements that might flow from that analysis.

A set of guidewords is used to prompt the consideration of possible failure modes. The guidewords used could be the classical FFA guidewords; not provided when required, provided when not required and incorrect operation. However, a better characterisation is the five guideword set put forward by Pumfrey in the SHARD method [Ref 24]. In this method, the guidewords are *omission*, *commission*, *early*, *late* and *value failure*.

As an example take the function provision of "secure and timely data flow" to and from an application process. One possible failure mode prompted by the "omission" guideword could be that the data is not sent from the source. There are a number of possible causes for this. First, the source partition (process) might not have been initialised within the system (this is a failure of the initialisation service). Another possible cause is that the process could not be scheduled (a failure of the initialisation and/or scheduling services). Yet a further cause could be that the source process may have terminated abnormally, which would be an application rather than an operating system failure.

If the timely data flow is required from an I/O device, failure of the device itself or failure of the inter-partition communication service (at a lower level, failure of the device driver) would be possible failure modes.

Once the possible failure modes and their causes have been identified in this way a set of derived requirements can be generated, and responsibility for carrying out each requirement assigned. For instance, non-scheduling of a sending process could be caught by means of a watchdog timer. As noted above, there is no direct provision in Linux for watchdog timers which are independent of the application processes. As a result this would become the responsibility of the application designers or the extended system environment, for example by designing a hardware watchdog timer card which detects failures of processes to provide signals in the correct sequence and at the correct intervals. Such a device is already used in a railway signalling application based upon a COTS operating system. The response to detection of this failure mode could also be put forward as a derived requirement on the application or system designers. For instance, backup functionality could be provided by means of a separate computer system.

Some work on the failure modes and derived safety requirements for the ARINC 653 model has already been undertaken by Conmy [Ref 25]. Note this analysis mainly places derived requirements on the operating system builder as the work assumes that a bespoke operating system is to be built. However, it shows that this approach to analysing the failure modes of an operating system is practical.

6.4 SUMMARY

This section has introduced a systematic method of classifying and analysing the characteristics of an operating system and deciding how well it will support the requirements of a safety related system.

The classification of operating system characteristics has three levels:

i) the highest level recognises six operating system FUNCTIONS which are in general required for a safety related system;

ii) the second level recognises 12 operating system SERVICES, each of which contributes to more than one function; and

iii) the third level is the API (or other means) by which the services are invoked by the application programs.

This classification scheme can be used for three purposes:

i) to decide whether a given operating system is intrinsically suitable for use with a given safety related application (in other words, whether it satisfies criterion C2);

ii) to compare the merits and disadvantages of a number of operating systems which are being considered for use in a safety related application; and

iii) to facilitate a hazard analysis of the interaction between the application programs and the operating system, to ensure that no new hazards have been introduced by the use of the operating system, and where necessary to create derived requirements to mitigate operating system failures or weaknesses.

7 LINUX CONFIGURATIONS AND APPLICATION EXAMPLES

In common with many general-purpose operating systems, Linux may be used in a number of configurations. These configurations are outlined in the following sections to illustrate the flexibility of the Linux system.

The hardware available to run a general-purpose operating system, such as Linux, ranges from the commonly available to the extremes of specialist interface cards. The original IBM Personal Computer (PC) comprised a motherboard that contained the memory, CPU and a number of interface slots into which additional interface cards could be installed. Typically these interface cards provided (colour or graphics) screen capabilities or interfaces to disk drives. Although technology has taken great steps since the early days of the IBM PC, the general arrangement of the hardware remains constant, if more highly integrated today. The motherboard with its interface slots is still the dominant form factor. As noted previously, this report is mainly concerned with “vanilla” hardware. The term “vanilla” is intended to describe commonly available interfaces, technologies and protocols in high volume use. One such example would be 10MHz or 100MHz Ethernet cards, which are widely established and commonly available, often at surprisingly low prices.

The more “flavoured” and unusual the hardware, technology, or protocol, the less likelihood there is of a substantial user base and hence the higher the probability of un-revealed errors in the hardware and software drivers. A number of Linux distributions for example SuSe [Ref 26] publish compatible hardware lists, which are worth consulting before purchase of the intended operational hardware,

Industrial environments are typically more demanding than the office desktop either due to environmental conditions, such as vibration, or EMC and EMI requirements. A number of vendors offer rack mounted computers either where many computers are required to occupy a small space or where harsh operational conditions demand rugged enclosures.

Linux systems will require the attention of a system administrator. This administration may be either local via a physically connected keyboard and screen or, where the system safety requirements permit, via some remote connection. At its most routine, system administration for a Linux system may consist of the inspection of the system log files, archiving and removal of old log information and the addition or replacement of hardware via planned maintenance. Support for the system administration function is provided through a number of documents either created through the LDP or published through third parties.

7.1 MODULAR KERNEL

The Linux kernel has a dynamic kernel module load facility to load additional functionality into the kernel on demand (in other words, as application programs require it). This facility is typically used to reduce the initial kernel size and to trade off the flexibility of the increased kernel functionality with speed. It is the normal mode of operation for “out of the box” Linux distributions.

Kernel modules are stored in a known reserved location on disk, and checked for availability as the kernel boots. Enabling the kernel checking of the module version number, before the module is loaded into the kernel, provides additional security.

7.2 MONOLITHIC KERNEL

In the monolithic kernel case, all facilities which may be used to dynamically enhance the kernel features are disabled. In the context of a safety related system a single monolithic kernel would be created (by means of the configuration mechanism supplied with the Linux distribution) which contained only that kernel functionality required to execute the applications required for the safety related system.

Additional measures are required to disable some of the network-based features by editing of the text files that describe the network services provided by the Internet Daemon inetd (/etc/inetd.conf and /etc/services).

Use of a monolithic kernel would provide a good argument that no unused code was loaded into the operating system.

Monolithic kernels may also be used in applications where computational resources are low, such as embedded systems. The embedded Linux kernel is available to run on platforms which do not use a Memory Management Unit (MMU) to implement virtual memory. In these circumstances additional code is required in the kernel to ensure that programs do not corrupt each other's memory, although the protection offered in such cases cannot be complete. As with specialised real-time Linux modifications, embedded Linux is not considered further in this report, but further information can be found, for example from the Embedded Linux Kernel Subset (ELKS) Project [Ref 27].

7.3 NETWORKED SYSTEMS

Networked systems share information across a network. Linux kernels, which support one or more networks, may be either monolithic or modular as the network drivers may either be embedded in the kernel or available as loadable modules. The most commonly used network protocol in the Unix and Linux world is the IP suite (including TCP/IP, UDP/IP, SNMP and ICMP), although Linux supports many other protocols.

Networked systems may be classified as either open or closed systems. Open systems are those systems that freely provide services to those who request the service (typically using the Internet). Measures to reduce the potential for abuse of the system require some form of access controls. A closed system should respond only to service requests from known sources. Both open and closed systems will demonstrate significantly reduced performance when a persistent and repeated request for a service cannot be satisfied. An example of such a repeated service request would be a denial of service security attack. A common defence in such circumstances is to provide services from behind a firewall or other network traffic filter to increase the overall availability of the system.

Common practice then, is to provide several layers of networked infrastructure to increase the dependability of the system. Routers, gateways and firewalls may provide the first layer of defence and be implemented by Linux (or other) systems. DNS, printing, mail and networked file system servers may be provided as the intermediate network infrastructure layer. The application layer then provides services to the applications on each server.

For a safety related system, hazards arising from open networks must be identified and controlled. Many safety related networked systems are electrically isolated from the outside world, or use dedicated links to outside equipment where interference from unauthorised persons and systems is not feasible. In such systems, hazards from external interference simply cannot arise or are extremely implausible. Increasingly, however, there are initiatives to connect safety related systems to the Internet and in such cases a suitable system architecture must be defined and analysed.

Figure 2 below illustrates the division between infrastructure, services and application layer of network systems.

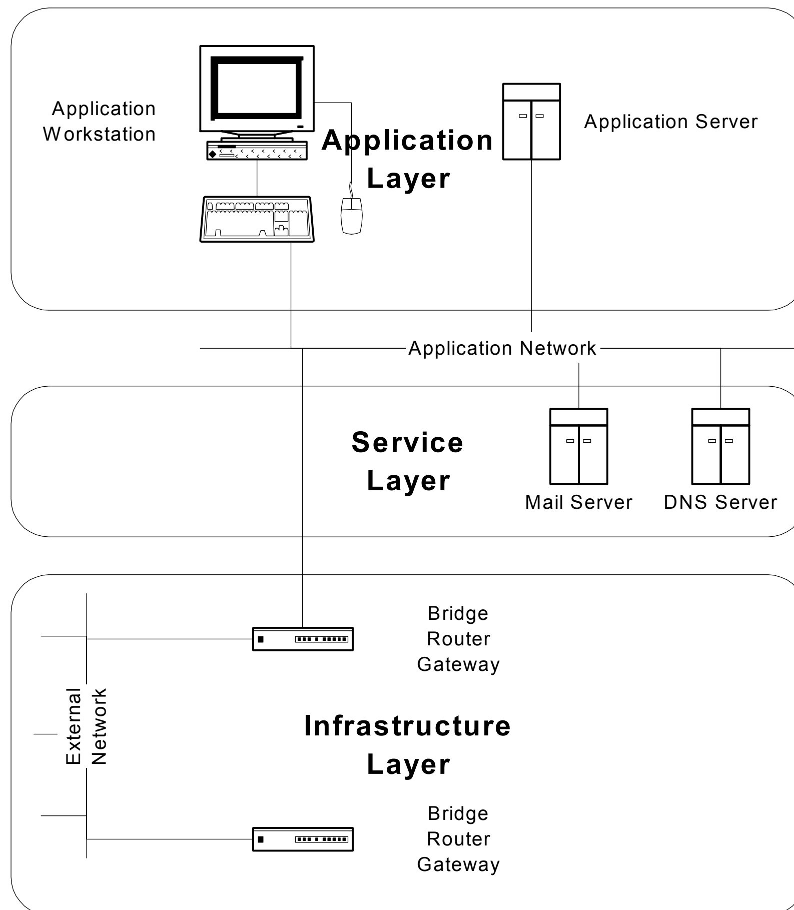


Figure 2 Typical network architecture for Linux systems

7.4 DISTRIBUTED SYSTEM(S)

Distributed systems may implement the safety related system as a suite of applications executed on a number of Linux servers. Typically these servers would communicate over a network (see above). Distributed systems need to manage the currency and timeliness of data passed between the computers involved.

Safety related distributed systems will require specific design elements to ensure the timely, error free and correct sequence delivery of data within the distributed system. Linux, indeed any general-purpose operating system, does not provide such data delivery integrity functions as standard. The role of the operating system in such safety related distributed systems is as a platform upon which the design elements could be based, together with the mitigations identified through safety analysis of the proposed safety related system.

Many of the application examples given in section 7.8 below are networked, distributed systems.

7.5 EMULATED (REPLACEMENT OR UPGRADED) SYSTEMS

Many Linux distributions provide emulation for older operating systems to run under Linux, for example the operating system DOSEmu emulates the PC DOS operating system.

When the original hardware platform for the safety related system becomes unreliable through the failing hardware, one possible solution might be to run the application (and its operating system) under Linux. The benefit of this solution is that failure of the application under the emulator can be reported to the system supervisor using the Linux features. At least one example of a safety related system using emulation in this way is known, running under a proprietary Unix.

7.6 REAL TIME LINUX

A number of suppliers offer so-called “real time Linux” systems. These fall into two classes:

i) systems where the standard Linux kernel is modified to provide behaviour more suitable for real time systems; and

ii) systems where the Linux kernel and all its application processes run as the lowest priority process of another real time kernel.

In the second case, the standard interrupt handling mechanism in the Linux kernel is modified to ensure that the underlying real time kernel can deal with interrupts properly.

There are some disadvantages in using such systems for safety related applications, among which are the following:

i) they are non-standard items and the amount of field service experience and other safety evidence available is likely to be substantially smaller than that of standard Linux distributions; and

ii) in the second case, the real time kernel must be certified in addition to the Linux system itself.

Like the embedded Linux variants, these real time Linux systems would not seem therefore to be suitable subjects for a certification project, although they may be of use in particular applications.

7.7 PLATFORMS

Linux is available for a wide variety of platforms. Originally developed for an Intel 80386 based system, Linux has been ported an extensive series of platforms from embedded small-resource system to large IBM mainframes.

In terms of the number of installed systems the IBM PC Intel based architecture predominates and much greater field service experience is available for Linux on such systems.

7.8 TYPICAL APPLICATIONS FOR WHICH LINUX MAY BE USEFUL

The following is a list of some safety related application types, some of which are currently running on Unix systems, for which Linux might be proposed.

i) ATC display systems (providing aircraft surveillance and flight plan data). As a general rule, these provide SIL 2 functions.

ii) railway control systems (provision of surveillance displays, automatic and manual route setting, control of individual devices, alarm handling, voice communication set-up). Railway control systems (as distinct from SIL 4 interlockings) are generally regarded as providing at most SIL 2 functions. Linux is currently being considered as the platform for a future UK railway control system;

iii) SCADA systems for railway electrification monitoring and control (SIL 1); and

iv) process plant display and control systems for plants in the oil and gas, chemical and water supply industries, generally of SIL 1 and SIL 2.

Note that none of these systems are embedded systems in the general sense of the word. Typical characteristics which these systems have in common are:

i) relatively modest timing requirements (responses to inputs are required in a few hundred milliseconds to a few seconds, rather than in the order of 10 milliseconds);

ii) the provision of high-resolution screen displays; and

iii) the need for disc file support.

Multi-screen colour displays supported either by a single or multiple graphics cards are also characteristic of these applications. They are also typically networked, distributed systems.

7.8.1 Case study of ATM system

In practice, many Unix systems have been successfully used to provide safety related functions which have response time requirements of the order of 0.5s on modern high speed processors.

A case in point is a civil radar display system where each display console receives about 100 radar track updates every second, and should update the corresponding objects on the screen with an ideal delay of no more than 250ms from receipt of the radar data from the network. This is not a hard real time safety requirement, however, as a longer delay of up to a second is acceptable on some occasions (perhaps due to user input requests) since each additional delay of 1s leads to a further discrepancy of around 0.1 nautical miles between the plotted position and true position of the aircraft, which is small by comparison with the standard horizontal separation of 5 nautical miles. The system in question also monitors the system CPU load and disables some computationally intensive actions in high load situations. Air traffic controllers using the system are also trained not to create input demands (such as repeated panning of the screen) which can result in consuming large amounts of processor time and causing the ATC situation display to lag behind reality to a dangerous degree.

Provided that the CPU loading is not close to 100% on a routine basis there is no reason to doubt that Linux could be used for similar systems.

8 LINUX RELEASE AND CONFIGURATION MECHANISMS

Linux Distributions are commonly packaged with a small amount of proprietary software. This proprietary software usually provides some added value to the user in the form a suite of applications to aid installation and system administration. These system administration tools are particularly important if Linux is to reach out successfully to a large number of desktop-based users, who typically will not be conversant with Linux and its command line utilities.

A good example of such a utility is the tool provided by the Red Hat company originally intended for their distribution and now in common use by the wider Linux community known as Red Hat Package Management (RPM) tool. The RPM tool combines a file compression utility and database to allow the installation of a software package and establish a database of dependencies between the package and other packages.

Linux distributions may be grouped as either horizontal or vertical. A horizontal distribution takes a selection of applications, libraries and utilities from across the Linux community and presents them as a representative snapshot. Examples of horizontal distributions are SuSe and Red Hat. A vertical distribution is a distribution that builds the entire selection of applications, libraries and utilities from scratch using the source code of the compiler as the starting point. A vertical distribution is usually more limited in that only those applications, libraries and utilities, which are available as source code, are provided as part of the vertical distribution.

The Linux kernel version is identified through a numbering system. Each kernel may be identified from the command-line via the command *'uname –a'* which requests that the kernel version build data and version (along with other information) be displayed. Numbers such as 1.2 or 2.2 identify the major releases with a sub-numbering system to identify the minor releases (for example, 2.2.19). If the last digit of the major release number is even, this identifies a general release which is considered to be stable enough for wider use. Odd numbered digits in the second place (for example, 1.3, 2.1) are considered to be development releases and as such are relatively unstable and unsuitable for general release.

Linux distributions continue to evolve. The Linux kernel continues to be developed although the rate of addition of new features has begun to slow over the past two years.

9 STRUCTURE OF THE LINUX SYSTEM

The Linux kernel is structured around a number of interfaces to the user and kernel services. The user is presented with an API that is embodied within user interfaces. The simplest user interface takes the form of a character based command shell such as *bash*. The bash shell is the re-implementation and improved version of the Unix Bourne Shell. Graphical User Interfaces are provided through windows managers. KDE is an example of a window manager provided with the SuSe Linux Distribution, and as noted in section 2 the X Windows system is available with Linux.

The kernel has a number of abstract layers that represent the interfaces to system services. For example, IDE hard drives require a physical driver, which is then utilised by a filesystem interface and finally presented to the kernel through a virtual file system interface. Other devices such as the interface to network cards are managed in the same way.

Figure 3 is a much-simplified illustration of the Linux kernel.

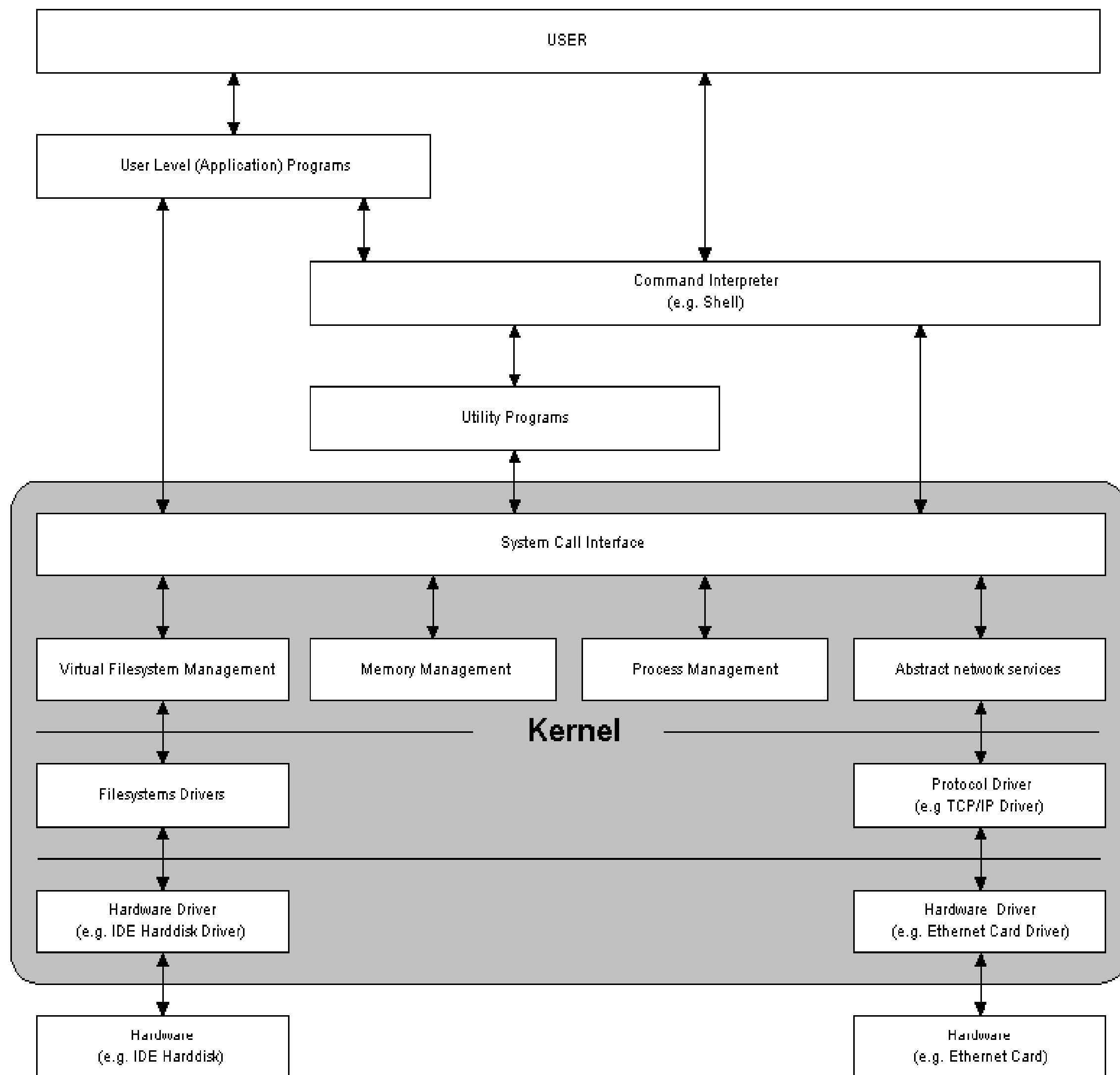


Figure 3 Simplified view of Linux system structure

The process management component is concerned with creating and switching user processes and with the management of internal kernel queues which are required to implement deferred operations, such as those which occur when the kernel has to wait for the completion of an I/O transfer or a timer expiry to fulfil a user service request.

Care has been taken in the Linux kernel to use abstract interfaces for the sake of clarity and portability. For example, there is a unified memory management model within the kernel which is used successfully to disguise the differences in memory management hardware between different processor architectures, such as Intel, Sparc and Compaq Alpha. Similarly, the virtual filesystem concept allows a number of specific filesystems to coexist within one Linux system.

The interface to the hardware is either by assembly code routines or assembly code embedded in C functions.

10 SOURCES OF EVIDENCE FOR SAFETY INTEGRITY

This section addresses criterion C3 introduced in section 3 above, namely the safety integrity which can be claimed for Linux. In SW01, three sources of evidence that software meets its safety requirement are recognised, namely:

i) field service experience;

ii) testing; and

iii) analysis.

As has been noted in section 3, specific safety requirements for an operating system cannot be derived, so the SW01 evidence requirements must be considered as requirements for evidence of integrity. The following three sections consider each source of evidence in turn.

It was hoped that the study would be able to obtain information from IBM on its use of Linux. Unfortunately, although IBM expressed a willingness to hold discussions on this matter, it was not possible to make suitable arrangements during the time period of the study, although a later project might be able to take advantage of cooperation with IBM. The information that would have been sought from IBM would have been the following:

i) arrangements for testing Linux on IBM platforms, including the extent and nature of the testing performed and an indication of the test results;

ii) the specification of Linux behaviour used by IBM to drive the testing process, and in particular if any specification document has been created;

iii) any inspections or other forms of analysis which IBM might have carried out on Linux; and

iv) any statistics available within IBM for the reliability of installed Linux systems.

10.1 FIELD SERVICE EXPERIENCE

There are a very large number of Linux systems in service. The Linux Counters project [Ref 28] is a collaborative project in which users register themselves with the project and their computers send usage statistics electronically. The number of users registered with the Counters project is some 95,000 with 125,000 individual computers. An estimate in [Ref 28] is that there may be 18 million users. A more conservative estimate indicates that there may be 7 or 8 million Linux systems in current service.

IBM, Sun Microsystems and other major hardware vendors offer Linux on their hardware. Many Internet Service Providers (ISPs) also use Linux, and a recent estimate gives Linux 27% of the ISP server market. Linux is also known to be used for the internal IT systems of some major organisations.

An argument of widespread use could therefore be made to justify the use of Linux in a safety related application [Ref 30]. However, it is rather more difficult to obtain good evidence of the true reliability of Linux in a variety of applications.

The Linux Counters project records the maximum "up time" of registered computers, and [Ref 28] shows a figure of 712 days as the largest "up time" encountered, with an average up time of 35 days. However, this "up time" figure is not a good estimate of MTBF for two reasons. Firstly, the "up time" figures are computed using an internal Linux value which wraps round on Intel processors so that one year is the maximum that is usually recorded. In addition, the reasons for "down time" are not recorded and many machines will be powered off for maintenance or kernel upgrades and the Linux systems on such machines will not have failed but will have been deliberately halted.

Experience of using Linux available within CSE indicates an MTBF of better than 30,000 hours and possibly approaching 50,000 hours.

One ISP reports no failures in a Linux server over a 10 month period during which monitoring was carried out, whereas by comparison a Windows NT server running the same application software and workload demonstrated an MTBF of six weeks.

The overall evidence of reliability from field service experience is therefore encouraging, but as a note of caution the following observations can be made:

i) the acceptable failure rates for ISP servers and other commercial operations are generally much lower than even the than SIL 1 requirement in IEC 61508, which is for an MTBF of 100,000 hours or better;

ii) reliability of the operating system has not traditionally been of great concern to hardware vendors, and

iii) tests show that even in mature operating systems failures can still occur when rarely used features are exercised [Ref 31].

Since Linux has been used much more extensively on IBM PC architecture computers than on any others, the field service experience is largely with this architecture.

10.1.1 Numeric reliability estimation

There is good evidence from the change history of Linux releases [Ref 14] that reliability growth has occurred with Linux due to the large numbers of users reporting problems and the fact that problems where reported are generally fixed quickly (and the fixes do not seem to cause problems elsewhere). The V2 series kernels are regarded generally as much more reliable than earlier V1 releases of the kernel.

The Adelard reliability prediction model in [Ref 30], which is based only upon the operating time to date and the number of errors remaining in the software under consideration, may give some indication of the expected reliability of Linux. As with any such model, the problem is deciding how many errors exist without knowing where the errors are (if they were known then presumably they would be fixed). The change records on the Linux HQ Web site [Ref 14] indicate that the various versions of the 2.4 kernel contain approximately 60 fixes for core operating system problems (this figure excludes corrections to a journalling filesystem, changes for new device drivers, or corrections to unusual device drivers). This figure can be rounded up to 100 for ease of calculation and to account for latent defects. On the assumptions that these defects existed in the Version 2.2 kernel and that this kernel was used for one year by 20,000 users on Intel (IBM PC) architecture computers, figures which are highly conservative, the Adelard reliability model predicts an MTBF well in excess of 1 million hours for systems using "vanilla" hardware (see section 7). Such a failure rate is in the SIL 2 range in IEC 61508.

Although the above is a very approximate calculation, it does support other evidence given above that Linux is highly reliable, provided of course that application programs themselves do not make erroneous API calls.

10.2 TESTING

Testing an operating system, especially the kernel, is different from application testing for a number of reasons:

i) there is no direct interaction with the user: problems have at least one level of indirection;

ii) operating systems are extremely "stateful", there being no "reset to known state" until reboot;

iii) hardware-dependence and ambience-dependence of errors means that small physical differences may hide a problem temporarily;

iv) high rate of changes;

v) radically different usage patterns for different users, for example desktop compared to file server; and

vi) automated testing tool support such as coverage analysis can be highly intrusive at the kernel level.

All these factors make it difficult to find repeatable, single-problem test cases. The alternative is to run an expected application workload against the kernel, with small variations (for example, different overload situations). Together with the results of general stress-tests these workloads convey a general impression of the overall stability of the operating system. This is the approach taken by the CERN Linux user group [Ref 32] and would be suitable for an individual project which is testing new releases of Linux for a safety related application.

There are some specific Linux test initiatives, as described below.

i) The Linux Test Project (LTP) [Ref 33], which is a consortium including major hardware vendors, delivers validation and robustness statements and verifies defect fixes through focus testing. Tests include regression, sanity and stress tests, endurance and performance runs, network tests (including TCP/IP) and filesystem tests. There are some 550 individual tests. Clearly, the work of the LTP provides some direct evidence that Linux has been subject to systematic testing and has passed the tests successfully. The tests in the LTP suite could form the basis of a safety certification test suite.

ii) The LSB team has also produced a suite to test against the LSB definition of Linux facilities. This suite could also be used to augment the LTP tests, although the study has not determined the degree of duplication or the consistency of these two test suites.

iii) The BALLISTA project at CMU has developed a robustness test suite for POSIX API calls, which has been used to check the robustness of Linux against calls with incorrect parameters or parameter values close to their extreme ranges. The results of this test are given in [Ref 20]. A program entitled "crashme" has also been developed which generates random process numbers and determines how the operating system responds to them, and it is reported that Linux survives this test better than most proprietary Unix systems.

One problem with the main LTP and LSB test suites is that they are not directly traceable to a specification of Linux behaviour, which would be desirable for a safety related certification project. 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.

The Linux kernel has a diagnostic output facility used for debugging and for error reporting in case of a kernel panic (the Linux term for an error of inconsistency which causes the kernel to abort and dump memory). It would therefore be feasible in principle to use this diagnostic output facility to carry out "dynamic analysis" of the Linux code. The code could be instrumented using well known test tools such as LDRA TestBed or Logiscope, and appropriate test coverage measurements including statement and branch coverage could be collected. Since the instrumentation would adversely affect performance, time critical sections of code such as interrupt handlers could not of course be instrumented. Test coverage measurement is highly recommended for higher SILs in IEC 61508 and other standards and would be a useful technique for building confidence in the integrity of Linux. SW01 requires evidence of a "high degree of test coverage".

To summarise, a test suite for use in a Linux certification project could be generated relatively easily by:

i) providing a written specification of Linux behaviour covering all the relevant SW01 attributes;

ii) reviewing and amalgamating existing test suites and supplementing them where required for untested areas and attributes; and

iii) providing traceability of the tests to the specification.

It would be feasible to make test coverage measurements and extend the test suite to obtain a higher degree of test coverage where necessary. Since instrumenting the code to obtain test coverage measurements could affect the behaviour of Linux, the tests should be repeated with both instrumented and non-instrumented code to ensure that the same results are obtained.

It is important to note that the written specification of Linux would be created solely for the purpose of capturing the intended behaviour of the existing system, not to specify any different behaviour. Any apparent defects would be communicated to the Linux maintainers for adjudication.

10.3 ANALYSIS

The term "analysis" in the software context means determining the properties of a software system, including behavioural properties, without actually executing it.

Analysis covers a wide range of techniques including, but not limited to, the following:

i) manual inspection of design and code for correctness and other qualities;

ii) code complexity measurements;

iii) checking conformance to coding standards for reliable software;

iv) control and dataflow analysis (which aims to find anomalous code);

v) semantic analysis (symbolic execution) which aims to state the behaviour of a software component by deriving expressions for the outputs in terms of the inputs;

vi) exception detection, which aims to determine which parts of a program cannot, may or will raise run-time exceptions such as numeric overflow, divide by zero and illegal address conditions;

vii) compliance analysis (formal proof of correctness against a specification); and

viii) worst case execution time analysis of object code.

The non-manual forms of analysis are often referred to a static analysis.

During the study, samples of the source code of the Linux kernel were subject to a brief manual inspection, and complexity analysis was carried out on the complete source tree using a shareware tool. Appendix C gives the output from the complexity analysis tool for the Intel version of the kernel (the complete results from the complexity analysis tool are available but are too bulky to include in this report). The conclusion of these investigations is that the kernel code is reasonably well commented and laid out and is fairly low in structural complexity on average, although some individual routines show high complexity which would need to be investigated by means of manual inspection. However, the code does not in any way conform to guidelines for C programming for safety related systems (such as the MISRA guidelines [Ref 34]).

Although some divergences from C coding rules are understandable, and others inevitable, in an operating system, most of them merely reflect the C style preferences of the originators for compactness over readability. Since subtle errors can arise from a number of the C constructs used in the kernel and elsewhere, a coding standard analyser such as QA-C could be used to highlight difficult areas for manual examination.

It is also clear that many of the more powerful forms of static analysis such as data flow analysis could not be attempted since existing tools such as MALPAS would not accept the Linux C code. The PolySpace tool, which performs exception detection, should be able to analyse the code since the suppliers claim that it accepts any ANSI C code.

It has been recommended in various places that Linux should be analysed for security defects, but the study was not able to find any evidence that this has been attempted. Moreover, security defects may not imply safety defects.

A project to certify Linux for safety related systems could therefore target manual inspection on:

i) any areas of high complexity as shown by code metrics;

ii) statements which do not conform to the MISRA C guidelines;

iii) code which may raise exceptions as shown by the PolySpace analyser; and

iv) hazard analysis, for example focusing on ill defined areas of behaviour or device drivers which might damage user space.

Inspection as described above would give further confidence that Linux is suitable for safety related applications. Inspection is highly recommended for SIL 3.

11 METHODOLOGY FOR LINUX CERTIFICATION

This section considers how Linux could be certified for use with safety related applications, in the event that this is perceived to be desirable by the project sponsors. A certification exercise would be most useful if it is expected that Linux would be used for SIL 3 applications. Even for lower integrity applications, certification would provide application developers and regulators with additional confidence in the integrity of Linux.

Since Linux runs on many platforms and is rapidly evolving, it would be desirable to have a repeatable certification process which could be used by regulators and others on new platforms and new releases.

Any certification materials and results should be made freely available in keeping with the open source philosophy.

Based on the material from previous sections, the certification project would in outline proceed as follows:

i) creation of a single specification of expected operating system behaviour, in as formal a notation as reasonably practicable given that a wide audience is expected. The specification should address all the SW01 behavioural attributes and would be intended to systematise and clarify existing definitions of Linux behaviour. It would be based on existing definitions such as POSIX with extensions where necessary;

ii) creation of a set of consolidated tests traceable to the specification;

iii) initial running of the test set to validate it and make corrections where necessary;

iv) inspection and analysis of the code as described in section 10; and

v) running of the test set with code instrumented to collect coverage measurements, and extension of the tests as required to increase coverage to the maximum extent which is reasonably practicable. As noted previously, the final test set should be run on both instrumented and non-instrumented versions of the code and the results compared.

The output of the initial certification project would be as follows:

i) a certification manual or handbook describing the process and materials used;

ii) the operating system specification document;

iii) the test suite;

iv) test results and test summary reports;

v) analysis and inspection reports, including recommendations for code changes where defects have been discovered; and

vi) a certification report identifying the hardware and software configuration being subjected to certification and summarising the conclusions of the certification exercise.

An important part of the certification report would be a list of known defects and anomalies to inform users of the system what should be avoided in use. If apparent software errors are found, these should also be listed in the report and made known to the Linux developers.

Clearly the first certification project would be the most costly since the certification process would have to be defined and the supporting materials produced. In addition, the analysis part of the certification would be largely one-off because, following initial certification, it would subsequently only be necessary to subject new or changed code to analysis.

Incremental certification of subsequent operating system releases could be achieved by analysing the changes and repeating inspections and tests of the changed features, together with overall validation testing.

New device drivers could be certified without a complete re-certification of the complete operating system.

The first certification exercise should be conducted on a platform which is likely to be widely used, such as an Intel Pentium based processor with standard peripherals such as serial and parallel interfaces, Ethernet LAN cards, two hard discs, CD-ROM, floppy disc and a graphics card capable of supporting more than one display screen. This configuration is suggested since, as argued in section 7, many proposed applications of Linux would be likely to use such a configuration.

The Linux components to be included in the initial certification project would be as follows:

i) the kernel in its monolithic variant;

ii) at least one disc based file system; and

iii) device drivers (including network support) for the hardware devices in the initial certification platform.

The "bash" command line interpreter should also be included but this could be omitted from the initial certification project if cost savings are desired.

11.1 EFFORT ESTIMATES

A very rough estimate of the effort required to carry out the initial certification project as proposed above is some six to eight person-years, not including the effort needed to test and analyse "bash".

Testing and analysis activities could largely be carried out in parallel.

A team of around four or five engineers could be deployed, giving a project timescale of 18 months to two years.

11.2 CERTIFICATION OF OTHER OPERATING SYSTEMS

The process outlined above could be used to certify any operating system. It must be recognised however that differences would be inevitable with other operating systems. Some obvious differences might be as follows:

i) another operating system may have a detailed specification already in existence, so that creation of a specification may not be necessary;

ii) another operating system may have a GUI as an integral part of the system, so that this would in practice have to be included in the certification process; and

iii) if the source code of the operating system is not publicly available, manual inspection, and collection of test coverage measurements by instrumenting the code, would not be feasible without the co-operation of the operating system supplier.

Testing of a GUI could be a complex and time consuming task, which has been avoided in the plan for Linux given above since a GUI is built on top of Linux as an application and is not integral to the operating system. This should be borne in mind if any certification project is planned.

12 CONCLUSIONS

On the basis of evidence from widespread use, some numeric reliability data, observed reliability growth, the existence of test projects and the limited analysis carried out by this study, it is concluded that “vanilla” Linux would be broadly acceptable for use in safety related applications of SIL 1 and SIL 2 integrity. This statement must of course be qualified by stating that the hardware must be of suitable integrity and that the application requirements were matched by the facilities of Linux.

It may also be feasible to certify Linux for use in SIL 3 applications by the provision of some further evidence from testing and analysis. Certification in this way would also increase confidence in the use of Linux in appropriate SIL 1 and 2 applications.

It is unlikely that Linux would be useful for SIL 4 applications and it would not be reasonably practicable to provide evidence that it meets a SIL 4 integrity requirement.

Any safety justification for using Linux in a given application should include evidence that an analysis has been carried out to show that Linux is suitable for that application, and that suitable mitigation is in place for any hazards arising from operating system failure. The framework given in section 6 would form a good basis for such an analysis.

It would be useful to provide a single definition of the behaviour of Linux which is as precise as reasonably practicable. Such a specification would facilitate hazard analysis, form a traceable basis for a certification test suite, and act as a baseline for the specification of the behaviour of modified versions of Linux including specialised device drivers and kernel scheduling modifications.

13 RECOMMENDATIONS

On the basis of the conclusions above, it is recommended that the project sponsors should consider the funding of a project to certify Linux to SIL 3 using the approach described in section 11 of this report.

It is also recommended that HSE and other regulators should look favourably on the use of Linux in SIL 1 and SIL 2 applications which are offered for safety approval, provided that appropriate safety cases are submitted covering all the issues raised in this report, for example criteria C1 and C2 and the issues of hazard analysis and derived requirements covered in section 6.

14 REFERENCES

Within the Linux community, the Internet is the preferred medium for the dissemination of Linux information. Accordingly, many of the references in this list are to World Wide Web (WWW) sites.

1 Requirements for the Use of Operating Systems in Safety Related Systems, report OS01, Parts 1 and 2, Civil Aviation Authority, Safety Regulation Group, Gatwick, March 1998 (unpublished).

2 Air Traffic Services Safety Requirement, Civil Aviation Authority, Safety Regulation Group, Gatwick, Document CAP 670 section SW01 "Regulatory Objective for Software in Safety Related Air Traffic Services", Issue 5, October 2001.

3 Pumfrey, D.J., (1999), The Principled Design of Computer System Safety Analyses, DPhil Thesis, University of York.

4 BS EN 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems, IEC, 2000.

5 DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc, 1992.

6 Taft, S. T et al (Eds), Consolidated Ada Reference Manual (ISO/IEC 8652 1995) with Technical Corrigendum 1, Lecture Notes in Computer Science 2219, Berlin: Springer 2001.

7 Information Technology – Portable Operating System Interface (Posix ®) – Part 1: System Application Program Interface (API) [C Language], ISO/IEC 9945-1, ANSI/IEEE Std 1003.1, 2nd Edition, 1996-07-12.

8 M. Beck et al, Linux Kernel Internals, Addison Wesley, 1996.

9 D P. Bovet & M Cesati, Understanding the Linux Kernel, O'Reilly & Associates Inc, 2001.

10 http://www.linux.org/docs/index.html

11 http://www.lpi.org

12 http://www.linuxtraining.co.uk

13 http://www.linuxbase.org/test/pilot/

14 http://www.linuxhq.com

15 Unused

16 NASA Contractor Report CR-1999-209347, available at http://www.csl.sri.com/users/rushby/partitioning.html.

17 N C Audsley et al, Applying new scheduling theory to static priority pre-emptive scheduling, Software Engineering Journal Vol 8 No 5 pp 284-292, 1993.

18 D. B. Cotton, Is Linux ready for military applications, COTS Journal, April 2001.

19 D. Saulsbury, Can open-source real-time Linux match proprietary real-time operating systems for military applications, COTS Journal, April 2001.

20 Koopman P, Comparing the Robustness of POSIX Operating Systems, Proc. 29th Annual Symposium on Fault Tolerant Computer Systems, 18-19 June 1999, Madison, Wisconsin.

21 ARINC 653: Avionics Application Software Standard Interface (1997).

22 Conmy, P. and J. McDermid (2001). High Level Failure Analysis for Integrated Modular Avionics. 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Brisbane, Australia.

23 http://csl.cse.ucsc.edu/smart.shtml

24 McDermid, J A, Nicholson, M, Pumfrey, D J & Fenelon, P, (1995), Experience with the application of HAZOP to computer-based systems, COMPASS '95: Proceedings of the Tenth Annual Conference on Computer Assurance, Gaithersburg, MD, pp. 37-48, IEEE, ISBN 0-7803-2680-2.

25 Nicholson, M., P. Conmy, et al. (2000). Generating and maintaining a Safety Argument for Integrated Modular Systems. 5th Australian Workshop on Safety Critical Systems and Software, Institution of Engineers Australia, 21 Bedford Street, North Melbourne, Victoria, AUSTRALIA.

26 http://www.suse.de/en

27 ELKS Project, http://elks.ecs.soton.ac.uk

28 Linux Counters Project, http://www.counter.li.org

29 Unused

30 Contract Research Report 336/2001, Justifying the use of software of uncertain pedigree (SOUP) in safety related applications, prepared by Adelard for the Health and Safety Executive, HSE Books, ISBN 0-7176-2010-7, and Contract Research Report 337/2001, Methods for assessing the safety integrity of safety related software of uncertain pedigree (SOUP), prepared by Adelard for the Health and Safety Executive, HSE Books, ISBN 0-7176-2011-5.

31 Koopman, P et al, Comparing Operating Systems using Robustness Benchmarks, Carnegie-Mellon University, August 1997.

32 http://wwwinfo.cern.ch/pdp/as/papers/linux-testing/certification_policy/.

33 Linux Test Project, http://ltp.sourceforge.net

34 MISRA Guidelines for the use of the C language in vehicle-based software, Motor Industry Research Association, UK 1998.

APPENDIX A

LINUX API FUNCTIONS

A1 KERNEL SYSTEM CALLS

When a user process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. For example, on an Intel x86 architecture the system call is invoked by executing the "int $0x80" assembly language instruction, with a system call number in the eax register, and other parameters in the remaining registers. These other parameters may include addresses of memory blocks, which allows passing of data and further parameters. The v2.4 kernel contains just over 200 system calls; the i386 assembly language structure defining them is given in section A3 (the `sys_ni_syscall` entries are dummy entries for system calls that are not implemented).

Application programs rarely invoke system calls directly. Programming languages with low-level interfaces will typically provide "wrapper" functions, which appear to the programmer as normal functions in the language and which in turn invoke the system call.

In addition to (or in some cases instead of) the system call wrappers, a programming language will usually provide functions that relate to typical programming tasks, and which are implemented using one or many system calls. So, for example, a C function

```
int open(const char *path, int oflag, ...)
```

which opens a file for reading, writing, or both, may need to execute system calls to determine whether the file exists, to determine whether the application program is permitted the requested access, to make the file available to the application program, and so on. This is the level of specification of the POSIX standard, and is the level considered in this report. The POSIX standard defines functions using the C programming language, but equivalents are likely to be available in most, if not all, programming languages available for Linux. For example, in Ada the equivalent to the C open function is the standard Ada95 Sequential_IO library procedure

```
Open(File : in out File_Type; Mode : in File_Mode;
     Name : in String; Form : in String := "").
```

Some calls at the POSIX API level are serviced without the use of any system calls.

A2 THE POSIX API

Linux has claimed POSIX compliance since version 2.2. To some extent this is dependent on the C libraries supplied with the kernel. These libraries are typically those from the GNU gcc compiler. Compliance with the POSIX standard provides a degree of portability of applications between platforms.

The POSIX API specifies language specific services for the C programming language. These are not discussed in this report, as any services that relate to the kernel are likely to be supported by services that are not specific to the C language.

The POSIX API also specifies data interchange formats and message queue management routines, which are outside the scope of the kernel.

A2.1 INTERRUPT HANDLING

The Linux operating system handling of interrupts is platform dependent. On an i386 platform IRQ lines 0, 2 and 13 are dedicated to the timer, the slave 8259A PIC and the mathematics coprocessor respectively. The other IRQs can be allocated dynamically; when a device driver requires an interrupt it issues a `request_irq()` call, and issues a `free_irq()` call when it no longer requires the IRQ. Other interrupts are handled by redirecting via an interrupt table which is initialised at start-up and which may be subsequently changed (for example, if a driver is dynamically loaded).

A2.2 PRIMARY MEMORY MANAGEMENT (ALLOCATION AND PROTECTION)

The primary memory management takes place internally to the kernel, and is transparent to the user.

A2.3 VIRTUAL MEMORY SERVICES INCLUDING SWAPPING AND PAGING

The kernel assigns a virtual address space to each process, and manages the mapping of that address space to primary and secondary memory in a way that is generally transparent to the application. The application has some limited control over this, described in Table A1 Virtual memory management services. See also mmap, described under Peripheral device handling.

Table A1 Virtual memory management services

`mlock`	Disable swapping of a portion of a process's address space. The POSIX standard specifies an implementation-optional restriction on the addresses that can be locked, but Linux 2.4 does not impose this restriction.
`mlockall`	Disable and enable swapping of a process's memory space. This can optionally lock all future memory allocated by the process. If this eventually causes the locked memory to exceed the available memory then the behaviour is implementation defined. At Linux v2.4, if this is due to a system call then the call will fail with the result `ENOMEM`; if it is due to growing stack then the stack growth will be denied and a `SIGSEGV` signal will be raised.

`mprotect`	Change the access protection for a portion of the address space.
`msync`	Write any modified data in a portion of memory to the permanent storage associated with that memory, if any.
`munlock`	Enable swapping of a portion of a process's memory space, however many times it has been locked.
`munlockall`	Enable swapping of a processes memory space.

A2.4 PROGRAM LOADING

At startup, the kernel executes the program `sbin/init`, which is centrally configured via the `etc/inittab` file. This will typically start a login program. The login program will examine the password file to determine the default command shell for the user, and execute that shell as a user process. The shell will typically execute its own startup script, which may be used to automatically start an application program, or the user can start application programs from the shell command line. The shell starts the application programs using the process creation and scheduling facilities described below.

A2.5 PROCESS CREATION AND SCHEDULING

Process creation follows the usual Unix model, shown in Table A2 Process creation and execution.

Table A2 Process creation and execution

`exec` family	The exec family of functions replaces one process with another. For example: `int execl(const char *path, const char *arg, ...)` replaces the current process with the executable program pointed to by path, with the arguments arg. On success, the exec family of functions do not return (because the calling process no longer exists). On failure they return -1.
`fork`	Create a new process that is an exact copy of the calling process, with a new (unique) process ID. On success, the function returns 0 to the original process, and the new process ID to the new process. On failure, -1 is returned. A call to fork is typically followed by a conditional call to one of the exec family to replace one of the processes with a different process.

There is a family of functions available for determining the status of processes, shown in Table A3 Process status. Terminated processes are retained in the process list (as "zombie" processes) until one of these functions is called. Linux has a permanent background task interrogating the status of orphaned processes, which allows them to be removed from the task list.

Table A3 Process status

`wait` family	Return the status of child processes, for example: `pid_t waitpid(pid_t pid, int *stat_loc, int options)` returns (in `stat_loc`) the status of one or more child processes, depending on the pid and the options.

Table A4 Threads

`nanosleep`	Suspend execution of a thread for a specified time interval (or until the sleep is interrupted by an appropriate signal).
`pause`	Suspend thread indefinitely, pending a signal.
`pthread_atfork`	Register fork handlers for execution before and after a fork.
`pthread_attr_destroy`	Destroy the thread attributes object, possibly by setting it to an invalid value.
`pthread_attr_getdetachedstate`	Get the thread detached state defined in a thread attribute object. If a thread is detached it is an error to use the thread ID and its storage can be reclaimed when it terminates.
`pthread_attr_getinheritsched`	Get the schedule inheritance policy, which determines whether a thread inherits its scheduling policy from the creating thread or whether it uses a fixed policy.
`pthread_attr_getschedparam`	Get the `schedparam` attribute of a `pthread_attr_t` argument. The only standardised member of the result, and the only member defined under Linux 2.4, is the scheduling priority.
`pthread_attr_getschedpolicy`	Get individual attributes of the pthread scheduling policy.
`pthread_attr_getscope`	Get the pthread contention scope.
`pthread_attr_getstacksize`	Get the minimum stacksize for a thread.
`pthread_attr_init`	Initialise a thread attributes object to implementation-defined default values.
`pthread_attr_setdetachedstate`	Set the thread detached state defined in a thread attribute object. If a thread is detached it is an error to use the thread ID and its storage can be reclaimed when it terminates.
`pthread_attr_setinheritsched`	Set the schedule inheritance policy, which determines whether a thread inherits its scheduling policy from the creating thread or whether it uses a fixed policy.

pthread_attr_setschedparam	Set the schedparam attribute of a pthread_attr_t argument. The only standardised member of the result, and the only member defined under Linux 2.4, is the scheduling priority.
pthread_attr_setschedpolicy	Set individual attributes of the pthread scheduling policy.
pthread_attr_setscope	Set the pthread contention scope.
pthread_attr_setstacksize	Set the minimum stacksize for a thread.
pthread_cancel	Request that a thread be cancelled.
pthread_create	Create a new thread.
pthread_detach	Detach a thread.
pthread_equal	Compare the IDs of two pthreads for equality.
pthread_exit	Terminate the calling thread.
pthread_getschedparam	Get the scheduling policy and parameters of an individual thread.
pthread_getspecific	Get the content of a thread-specific data block.
pthread_getstackaddr	Set the address of the stack for a thread.
pthread_join	Suspend the calling thread until the target thread terminates.
pthread_key_create	Create a thread-specific data block. The key (returned as a parameter) used to access the data block will be available to all threads of a process, but the values associated with the key will be thread specific.
pthread_key_delete	Delete a thread-specific data block.
pthread_mutex_getpriorityceiling	Get the mutex priority ceiling.
pthread_mutex_setpriorityceiling	Set the mutex priority ceiling.
pthread_mutexattr family	Manipulate mutex attribute objects. This allows the priority of a thread to me modified according to the priority of threads waiting on the mutex.
pthread_once	The first call within a thread calls a specified function. Subsequent calls within the thread do not call the function.
pthread_self	Return the thread ID of the calling thread.
pthread_setcancelstate	Set the cancelability state of the calling thread.
pthread_setcanceltype	Set the cancelability type of the calling thread.
pthread_setschedparam	Set the scheduling policy and parameters of an individual thread.
pthread_setspecific	Set the content of a thread-specific data block.
pthread_setstackaddr	Set the address of the stack for a thread.
pthread_testcancel	Set a cancellation point in the calling thread.
sleep	Suspend thread for a specific period of time

Table A5 Scheduling

_exit	Terminate the calling process.
sched_get_priority_max	Get the maximum priority available under a scheduling policy.
sched_get_priority_min	Get the minimum priority available under a scheduling policy.
sched_getparam	Get the scheduling parameters of a process.
sched_getscheduler	Get the scheduling policy of a process.
sched_rr_get_interval	Get the current execution time limit for a process.
sched_setparam	Set the scheduling parameters of a process.
sched_setscheduler	Set the scheduling policy and parameters of a process.
sched_yield	Force the running thread to relinquish the processor until it again becomes head of the thread list.

A number of scheduling operations are described under Signals in Inter-process communication.

A2.6 SECONDARY STORAGE MANAGEMENT

The Linux kernel can manage a variety of different file systems simultaneously. The kernel API provides a Virtual File System (VFS) to make the differences in file system transparent to application programs. The VFS provides functionality intended both for application programs and for the interfaces to the actual file systems.

A Linux file can be one of the following types:

i) Regular file;

ii) Directory;

iii) Symbolic Link;

iv) Block-oriented device file;

v) Character-oriented device file;

vi) Pipe and named pipe; and

vii) Socket.

Types iv) and v) are means of peripheral device management, and are managed by device drivers. Types vi) and vii) are means of interprocess communication, and are discussed under that heading.

The VFS uses inode structures in a similar way to Unix. The inode is a structure defining a file (except for the filename which is stored in the directory entry). The VFS also uses Directory Entry (dentry) structure, an in-core structure defining a file's details: inode, parent dentry and so on.

See also `mmap` under Peripheral device handling.

Table A6 Directory operations

`chdir`	Change the current working directory.
`closedir`	Close the directory stream.
`getcwd`	Get the current working directory.
`mkdir`	Create a new directory.
`opendir`	Open a directory stream, initialised to the first file in the directory.
`readdir` family	Return a pointer to the current directory entry and advances to the next entry. It returns NULL if there are no more entries.
`rewinddir`	Reset the position to the first entry.
`rmdir`	Removes a link to a directory and decrements its reference count. When the reference count reaches zero the space occupied by the directory is freed for reuse.

Table A7 File operations

`access`	Identifies whether a particular file is accessible in a particular mode, optionally for a particular set of permissions.
`chmod` family	Change the file permissions.
`chown`	Changes the owner and group of a file.
`close`	Deallocates the file descriptor, making the file no longer available to the process through that descriptor.
`creat`	Create a new file (possibly overwriting an existing one); it is equivalent to `open(`*`path`*`, O_WRONLY | O_CREAT | O_TRUNC, `*`mode`*`)`.
`dup` family	Duplicate open file descriptors. They provide an alternative interface to the `fcntl` function.
`fcntl`	Provide a variety of interrogation and control functions for open files.
`fsync` family	Synchronise the internal representation of the state of a file with the representation on the storage medium.
`ftruncate`	Truncate a file to a specified length.
`link`	Create a hard link to an existing file.
`lseek`	Reposition the read/write file offset.

open	Make a file available for reading, writing or both.
pathconf family	Determines the value of a configurable limit or option associated with a file or directory. The POSIX standard identifies 12 such variables.
read	Attempt to read a number of bytes from a file into a buffer.
rename	Change the name of a file.
stat family	Return the status of a file.
umask	Set the file mode creation mask of the calling process, which has the effect of limiting the file permissions of the process.
unlink	Remove a link to a file and decrements the files reference count. When the reference count reaches zero the space occupied by the file is freed for reuse.
utime	Set the access and modification times of a file.
write	Attempt to write a number of bytes from a buffer to a file.

Table A8 Asynchronous file operations

aio_cancel	Attempt to cancel an asynchronous I/O operation.
aio_error	Interrogate the error status of a queued asynchronous file operation.
aio_fsync	Request an fsync operation and return immediately, whether or not the fsync operation has completed.
aio_read	Request a read operation and return immediately, whether or not the read operation has completed.
aio_return	Interrogate the return status of a queued asynchronous file operation.
aio_suspend	Suspend the calling thread until at least one of a list of asynchronous I/O operations completes.
aio_write	Request a write operation and return immediately, whether or not the write operation has completed.
lio_listio	Request a list of I/O operations with a single function call.

A2.7 INTER-PROCESS COMMUNICATION

A2.7.1 Signals

Signals are messages that can be sent to processes. Each process has a default response to each possible signal, which can be overridden. POSIX defines 20 signals.

Table A9 Signals

alarm	Generate a signal at a particular time.
kill	Send a message to a process, requesting that the process terminate.
pthread_kill	Send a message to a thread, requesting that the thread terminate.
pthread_sigmask	Examine and change blocked threads.
sigaction	Examines or specifies the response to signals.
sigpending	Examine pending signals.
sigprocmask	Examine and change blocked processes.
sigqueue	Queue a signal to a process.
sigsetops family	Manipulate sets of signals.
sigsuspend	Wait for a signal.
sigwait family	Synchronously accept signals.

A2.7.2 Pipes

A pipe is a pseudo-file that can be used to pass data between applications. A pipe is created with the function:

```
int pipe(int fildes[2])
```

The function inserts file descriptors into fildes[0] and fildes[1]. Data can be written to fildes[1] and read from fildes[0] on a first-in, first-out basis, using normal file operations.

A2.7.3 Named Pipes

Named pipes, or FIFOs, are pseudo-files, which can be written to and read from by any process. They are always written to at the end and read from at the beginning, so they work as First-In, First-Out queues.

FIFOs are created with the `mkfifo` function. Once created, they can be manipulated using the file `open`, `read`, `write` and `close` functions.

A2.7.4 Semaphores

Semaphores are counters associated with data blocks or resources. Semaphores can be shared between processes, and so communicate the availability of a data block or resource.

Table A10 Semaphores

sem_close	Close a named semaphore.
sem_destroy	Destroy an unnamed semaphore.
sem_getvalue	Get the value of a semaphore.

sem_init	Initialise an unnamed semaphore.
sem_open	Initialise/open a named semaphore.
sem_post	Unlock a semaphore.
sem_trywait	Lock semaphore. If the semaphore is already locked, sem_trywait does not block.
sem_unlink	Remove a named semaphore.
sem_wait	Lock semaphore. If the semaphore is already locked, sem_wait blocks until it is unlocked or the call is interrupted by a signal.

A2.7.5 Mutexes

A mutex provides a thread with exclusive access to a resource.

Table A11 Mutexes

pthread_mutex_lock	Lock a mutex. If the mutex is already locked, this blocks until it becomes available.
pthread_mutex_trylock	Lock a mutex. If the mutex is already locked, return failure.
pthread_mutex_unlock	Unlock a mutex.
pthread_mutexattr_destroy	Destroy a mutex attributes object.
pthread_mutexattr_getpshared	Get the behaviour of mutexes shared between processes.
pthread_mutexattr_init	Initialise a mutex attributes object with a set of default values.
pthread_mutexattr_setpshared	Set the behaviour of mutexes shared between processes.

A2.7.6 Conditions

Conditions (short for condition variables) are a synchronisation mechanism that allows threads to suspend execution until some predicate on shared data is satisfied. Conditions should be associated with mutexes, to avoid race conditions.

The Linux threads implementation, as at v2.4, does not support attributes on conditions. The relevant POSIX functions are implemented for POSIX compliance only and have no effect.

Table A12 Conditions

`pthread_cond_destroy`	Destroy a condition.
`pthread_cond_init`	Create a condition.
`pthread_cond_timedwait`	Block on a condition with timeout.
`pthread_cond_timedwait`	Raise an error if a specified time passes in the wait condition.
`pthread_cond_wait`	Block on a condition.

A2.7.7 Memory maps

A memory map is a mapping between a part of a process's address space and a memory object. This can optionally be shared between processes, and so allows a means of communication. It can also map a file to memory.

Table A13 Memory maps

`mmap`	Establish a memory map.
`munmap`	Removes a mapping.
`shm_open`	Associate a shared memory object and a file descriptor that can be used to access that shared memory object.
`shm_unlink`	Remove a shared memory object.

A2.8 PROVISION OF FILESTORE

The Linux kernel manages a virtual filestore described under Secondary Storage Management. Actual filestores are managed by device drivers external to the kernel.

A2.9 PERIPHERAL DEVICE HANDLING

Most peripheral device handling is done by means of device drivers that are added to the kernel. These drivers are beyond the scope of this report.

A2.9.1 Terminal interface

The terminal interface is supported on any asynchronous communications ports provided by the implementation. It is implementation defined whether it is supported for network connections or synchronous ports.

Terminal input can operate in one of two modes: canonical and non-canonical. Essentially, canonical mode is line-oriented and non-canonical is character oriented. This also affects the blocking behaviour of read operations.

For most purposes, a terminal interface appears to application software as a file, but it has some extra functions.

Table A14 Terminal interfaces

`cfspeed` family	Set and read the Baud rate of the terminal.
`tcsetattr`	Set the attributes in a `termios` structure based on the attributes associated with an open file descriptor.
`tcgetattr`	Get the attributes in a `termios` structure based on the attributes associated with an open file descriptor.
`tcsendbreak`	Cause transmission of a continuous stream of zero-valued bits for a period of time defined by a parameter.
`tcdrain`	Wait until all output associated with a terminal has been written.
`tcflush`	Discard all data received but not read, all data written but not sent, or both.
`tcflow`	Suspend or restart input or output flow on a terminal device.
`tcsetgrp`	Set the foreground process group ID.
`tcgetgrp`	Get the foreground process group ID.

A2.9.2 Sockets

Sockets are handled as a special type of character-oriented file device. They do not have inodes, and so are accessed through their own set of system calls. Sockets are identified but not specified by POSIX.

Table A15 Sockets

`socket`	Create a socket.
`bind`	Associate a socket with an address.
`listen`	Set the maximum number of incoming requests that will be queued before requests are denied.
`accept`	Accept a connection on a socket.
`connect`	Try to connect to a listening socket.

Once a connection is established, the normal file `read`, `write` and `close` functions are used to manage it.

A2.9.3 Memory maps

Memory maps, described under Peripheral Device Handling, can be used to interface with memory-mapped devices.

A2.10 OPERATOR AND USER INTERFACE HANDLING (character or graphical)

The operator and user interface handling is done either as a terminal interface as described under Peripheral Device Handling, or through device drivers supplemental to the kernel.

A2.11 NETWORK COMMUNICATIONS AND DISTRIBUTED FACILITIES

Network communications and distributed facilities is done either as a terminal interface as described under Peripheral Device Handling, or through device drivers supplemental to the kernel.

A2.12 ACCESS CONTROL

Table 16 Access control

`getegid`	Get the effective group ID.
`geteuid`	Get the effective user ID.
`getgid`	Get the group ID.
`getgroups`	Get supplementary group IDs.
`getlogin` family	Get the user name.
`getpgrp`	Get the process group ID.
`getuid`	Get the user ID.
`setgid`	Set the group ID.
`setpgid`	Set the process group ID for job control.
`setsid`	Create a session and sets the process group ID.
`setuid`	Set the user ID.

A2.13 ENVIRONMENT

The POSIX API makes available an array of strings, called “environ” (short for “environment”:

```
extern char **environ;
```

The use of these strings is generally application dependent, but there are standard entries defined to identify such things as standard file locations and internationalisation options. Specific entries in the environment can be retrieved using the `getenv` function.

The POSIX standard also specifies the presence of 36 constants in `limits.h`, which identify the most restrictive maximum and minimum values of standard types, and identifies a further 23 optional values and 16 specified by the C standard.

The POSIX standard specifies 24 compile-time and 6 execution-time symbolic constants that specify the system capabilities.

Table A17 Environment operations

ctermid	Identify the current controlling terminal for the current process.
getgrgid	Return a pointer to the gid group structure.
getgrnam	Return a pointer to the name group structure.
getpw family	Return information from the password table relating to a particular user ID or name. This information does not include the password itself.
sysconf	Get the current value of a configurable system limit or option. The POSIX standard identifies 52 such variables.
time	Get the system time in seconds since 1970-01-01 00:00:00+00.
times	Return time accounting information for the session.
ttyname	Identify the terminal associated with a particular file designator.
uname	Get the system name.

A2.14 CLOCKS AND TIMERS

Table A18 Clocks and timers

clock_settime	Set the time of a specified clock.
clock_gettime	Get the time of a specified clock.
clock_getres	Get the resolution of a specified clock.
timer_create	Create a per-process timers.
timer_delete	Delete a per-process timers.
timer_settime	Set the time until the next expiration of a timer.
timer_gettime	Get the time until the next expiration of a timer.
timer_getoverrun	When a timer expires it sends a signal to the process. Only one such signal can be pending for a process from each timer at any time. This function returns the number of subsequent expirations. This function is optional in POSIX, and not included in the “info” Linux documentation with Linux 2.4, but is defined in the C headers supplied with the SuSE 7.1 Linux 2.4 distribution.

A3 LIST OF SYSTEM CALLS

```
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall)          /* 0-old "setup()" system call*/
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open)                /* 5 */
    .long SYMBOL_NAME(sys_close)
    .long SYMBOL_NAME(sys_waitpid)
    .long SYMBOL_NAME(sys_creat)
    .long SYMBOL_NAME(sys_link)
    .long SYMBOL_NAME(sys_unlink)              /* 10 */
    .long SYMBOL_NAME(sys_execve)
    .long SYMBOL_NAME(sys_chdir)
    .long SYMBOL_NAME(sys_time)
    .long SYMBOL_NAME(sys_mknod)
    .long SYMBOL_NAME(sys_chmod)               /* 15 */
    .long SYMBOL_NAME(sys_lchown16)
    .long SYMBOL_NAME(sys_ni_syscall)          /* old break syscall holder */
    .long SYMBOL_NAME(sys_stat)
    .long SYMBOL_NAME(sys_lseek)
    .long SYMBOL_NAME(sys_getpid)              /* 20 */
    .long SYMBOL_NAME(sys_mount)
    .long SYMBOL_NAME(sys_oldumount)
    .long SYMBOL_NAME(sys_setuid16)
    .long SYMBOL_NAME(sys_getuid16)
    .long SYMBOL_NAME(sys_stime)               /* 25 */
    .long SYMBOL_NAME(sys_ptrace)
    .long SYMBOL_NAME(sys_alarm)
    .long SYMBOL_NAME(sys_fstat)
    .long SYMBOL_NAME(sys_pause)
    .long SYMBOL_NAME(sys_utime)               /* 30 */
    .long SYMBOL_NAME(sys_ni_syscall)          /* old stty syscall holder */
    .long SYMBOL_NAME(sys_ni_syscall)          /* old gtty syscall holder */
    .long SYMBOL_NAME(sys_access)
    .long SYMBOL_NAME(sys_nice)
    .long SYMBOL_NAME(sys_ni_syscall)          /* 35 */
    .long SYMBOL_NAME(sys_sync)
    .long SYMBOL_NAME(sys_kill)
    .long SYMBOL_NAME(sys_rename)
    .long SYMBOL_NAME(sys_mkdir)
    .long SYMBOL_NAME(sys_rmdir)               /* 40 */
    .long SYMBOL_NAME(sys_dup)
    .long SYMBOL_NAME(sys_pipe)
    .long SYMBOL_NAME(sys_times)
    .long SYMBOL_NAME(sys_ni_syscall)          /* old prof syscall holder */
    .long SYMBOL_NAME(sys_brk)                 /* 45 */
    .long SYMBOL_NAME(sys_setgid16)
```

```
.long SYMBOL_NAME(sys_getgid16)
.long SYMBOL_NAME(sys_signal)
.long SYMBOL_NAME(sys_geteuid16)
.long SYMBOL_NAME(sys_getegid16)          /* 50 */
.long SYMBOL_NAME(sys_acct)
.long SYMBOL_NAME(sys_umount)             /* recycled never used phys() */
.long SYMBOL_NAME(sys_ni_syscall)         /* old lock syscall holder */
.long SYMBOL_NAME(sys_ioctl)
.long SYMBOL_NAME(sys_fcntl)              /* 55 */
.long SYMBOL_NAME(sys_ni_syscall)         /* old mpx syscall holder */
.long SYMBOL_NAME(sys_setpgid)
.long SYMBOL_NAME(sys_ni_syscall)         /* old ulimit syscall holder */
.long SYMBOL_NAME(sys_olduname)
.long SYMBOL_NAME(sys_umask)              /* 60 */
.long SYMBOL_NAME(sys_chroot)
.long SYMBOL_NAME(sys_ustat)
.long SYMBOL_NAME(sys_dup2)
.long SYMBOL_NAME(sys_getppid)
.long SYMBOL_NAME(sys_getpgrp)            /* 65 */
.long SYMBOL_NAME(sys_setsid)
.long SYMBOL_NAME(sys_sigaction)
.long SYMBOL_NAME(sys_sgetmask)
.long SYMBOL_NAME(sys_ssetmask)
.long SYMBOL_NAME(sys_setreuid16)         /* 70 */
.long SYMBOL_NAME(sys_setregid16)
.long SYMBOL_NAME(sys_sigsuspend)
.long SYMBOL_NAME(sys_sigpending)
.long SYMBOL_NAME(sys_sethostname)
.long SYMBOL_NAME(sys_setrlimit)          /* 75 */
.long SYMBOL_NAME(sys_old_getrlimit)
.long SYMBOL_NAME(sys_getrusage)
.long SYMBOL_NAME(sys_gettimeofday)
.long SYMBOL_NAME(sys_settimeofday)
.long SYMBOL_NAME(sys_getgroups16)        /* 80 */
.long SYMBOL_NAME(sys_setgroups16)
.long SYMBOL_NAME(old_select)
.long SYMBOL_NAME(sys_symlink)
.long SYMBOL_NAME(sys_lstat)
.long SYMBOL_NAME(sys_readlink)           /* 85 */
.long SYMBOL_NAME(sys_uselib)
.long SYMBOL_NAME(sys_swapon)
.long SYMBOL_NAME(sys_reboot)
.long SYMBOL_NAME(old_readdir)
.long SYMBOL_NAME(old_mmap)               /* 90 */
.long SYMBOL_NAME(sys_munmap)
.long SYMBOL_NAME(sys_truncate)
.long SYMBOL_NAME(sys_ftruncate)
.long SYMBOL_NAME(sys_fchmod)
.long SYMBOL_NAME(sys_fchown16)           /* 95 */
.long SYMBOL_NAME(sys_getpriority)
```

```
.long SYMBOL_NAME(sys_setpriority)
.long SYMBOL_NAME(sys_ni_syscall)           /* old profil syscall holder */
.long SYMBOL_NAME(sys_statfs)
.long SYMBOL_NAME(sys_fstatfs)              /* 100 */
.long SYMBOL_NAME(sys_ioperm)
.long SYMBOL_NAME(sys_socketcall)
.long SYMBOL_NAME(sys_syslog)
.long SYMBOL_NAME(sys_setitimer)
.long SYMBOL_NAME(sys_getitimer)            /* 105 */
.long SYMBOL_NAME(sys_newstat)
.long SYMBOL_NAME(sys_newlstat)
.long SYMBOL_NAME(sys_newfstat)
.long SYMBOL_NAME(sys_uname)
.long SYMBOL_NAME(sys_iopl)                 /* 110 */
.long SYMBOL_NAME(sys_vhangup)
.long SYMBOL_NAME(sys_ni_syscall)           /* old "idle" system call */
.long SYMBOL_NAME(sys_vm86old)
.long SYMBOL_NAME(sys_wait4)
.long SYMBOL_NAME(sys_swapoff)              /* 115 */
.long SYMBOL_NAME(sys_sysinfo)
.long SYMBOL_NAME(sys_ipc)
.long SYMBOL_NAME(sys_fsync)
.long SYMBOL_NAME(sys_sigreturn)
.long SYMBOL_NAME(sys_clone)                /* 120 */
.long SYMBOL_NAME(sys_setdomainname)
.long SYMBOL_NAME(sys_newuname)
.long SYMBOL_NAME(sys_modify_ldt)
.long SYMBOL_NAME(sys_adjtimex)
.long SYMBOL_NAME(sys_mprotect)             /* 125 */
.long SYMBOL_NAME(sys_sigprocmask)
.long SYMBOL_NAME(sys_create_module)
.long SYMBOL_NAME(sys_init_module)
.long SYMBOL_NAME(sys_delete_module)
.long SYMBOL_NAME(sys_get_kernel_syms)      /* 130 */
.long SYMBOL_NAME(sys_quotactl)
.long SYMBOL_NAME(sys_getpgid)
.long SYMBOL_NAME(sys_fchdir)
.long SYMBOL_NAME(sys_bdflush)
.long SYMBOL_NAME(sys_sysfs)                /* 135 */
.long SYMBOL_NAME(sys_personality)
.long SYMBOL_NAME(sys_ni_syscall)           /* for afs_syscall */
.long SYMBOL_NAME(sys_setfsuid16)
.long SYMBOL_NAME(sys_setfsgid16)
.long SYMBOL_NAME(sys_llseek)               /* 140 */
.long SYMBOL_NAME(sys_getdents)
.long SYMBOL_NAME(sys_select)
.long SYMBOL_NAME(sys_flock)
.long SYMBOL_NAME(sys_msync)
.long SYMBOL_NAME(sys_readv)                /* 145 */
.long SYMBOL_NAME(sys_writev)
```

```
.long SYMBOL_NAME(sys_getsid)
.long SYMBOL_NAME(sys_fdatasync)
.long SYMBOL_NAME(sys_sysctl)
.long SYMBOL_NAME(sys_mlock)                    /* 150 */
.long SYMBOL_NAME(sys_munlock)
.long SYMBOL_NAME(sys_mlockall)
.long SYMBOL_NAME(sys_munlockall)
.long SYMBOL_NAME(sys_sched_setparam)
.long SYMBOL_NAME(sys_sched_getparam)           /* 155 */
.long SYMBOL_NAME(sys_sched_setscheduler)
.long SYMBOL_NAME(sys_sched_getscheduler)
.long SYMBOL_NAME(sys_sched_yield)
.long SYMBOL_NAME(sys_sched_get_priority_max)
.long SYMBOL_NAME(sys_sched_get_priority_min)   /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)
.long SYMBOL_NAME(sys_setresuid16)
.long SYMBOL_NAME(sys_getresuid16)              /* 165 */
.long SYMBOL_NAME(sys_vm86)
.long SYMBOL_NAME(sys_query_module)
.long SYMBOL_NAME(sys_poll)
.long SYMBOL_NAME(sys_nfsservctl)
.long SYMBOL_NAME(sys_setresgid16)              /* 170 */
.long SYMBOL_NAME(sys_getresgid16)
.long SYMBOL_NAME(sys_prctl)
.long SYMBOL_NAME(sys_rt_sigreturn)
.long SYMBOL_NAME(sys_rt_sigaction)
.long SYMBOL_NAME(sys_rt_sigprocmask)           /* 175 */
.long SYMBOL_NAME(sys_rt_sigpending)
.long SYMBOL_NAME(sys_rt_sigtimedwait)
.long SYMBOL_NAME(sys_rt_sigqueueinfo)
.long SYMBOL_NAME(sys_rt_sigsuspend)
.long SYMBOL_NAME(sys_pread)                    /* 180 */
.long SYMBOL_NAME(sys_pwrite)
.long SYMBOL_NAME(sys_chown16)
.long SYMBOL_NAME(sys_getcwd)
.long SYMBOL_NAME(sys_capget)
.long SYMBOL_NAME(sys_capset)                   /* 185 */
.long SYMBOL_NAME(sys_sigaltstack)
.long SYMBOL_NAME(sys_sendfile)
.long SYMBOL_NAME(sys_ni_syscall)               /* streams1 */
.long SYMBOL_NAME(sys_ni_syscall)               /* streams2 */
.long SYMBOL_NAME(sys_vfork)                    /* 190 */
.long SYMBOL_NAME(sys_getrlimit)
.long SYMBOL_NAME(sys_mmap2)
.long SYMBOL_NAME(sys_truncate64)
.long SYMBOL_NAME(sys_ftruncate64)
.long SYMBOL_NAME(sys_stat64)                   /* 195 */
.long SYMBOL_NAME(sys_lstat64)
```

```
.long SYMBOL_NAME(sys_fstat64)
.long SYMBOL_NAME(sys_lchown)
.long SYMBOL_NAME(sys_getuid)
.long SYMBOL_NAME(sys_getgid)                   /* 200 */
.long SYMBOL_NAME(sys_geteuid)
.long SYMBOL_NAME(sys_getegid)
.long SYMBOL_NAME(sys_setreuid)
.long SYMBOL_NAME(sys_setregid)
.long SYMBOL_NAME(sys_getgroups)                /* 205 */
.long SYMBOL_NAME(sys_setgroups)
.long SYMBOL_NAME(sys_fchown)
.long SYMBOL_NAME(sys_setresuid)
.long SYMBOL_NAME(sys_getresuid)
.long SYMBOL_NAME(sys_setresgid)                /* 210 */
.long SYMBOL_NAME(sys_getresgid)
.long SYMBOL_NAME(sys_chown)
.long SYMBOL_NAME(sys_setuid)
.long SYMBOL_NAME(sys_setgid)
.long SYMBOL_NAME(sys_setfsuid)                 /* 215 */
.long SYMBOL_NAME(sys_setfsgid)
.long SYMBOL_NAME(sys_pivot_root)
.long SYMBOL_NAME(sys_mincore)
.long SYMBOL_NAME(sys_madvise)
.long SYMBOL_NAME(sys_getdents64)               /* 220 */
.long SYMBOL_NAME(sys_fcntl64)
.long SYMBOL_NAME(sys_ni_syscall)               /* reserved for TUX */
```

APPENDIX B

MAPPING OF THE OPERATING SYSTEM SERVICE MODEL TO LINUX SYSTEM CALLS

This appendix provides a mapping between the classification of operating system services given in section 6 of the main report to the Linux V2.4 API as presented in Appendix A. Note that threads cannot be regarded as a partitioning mechanism, but can be seen as providing support for scheduling and intra-partition communication.

Linux: data loading:

Linux systems are started (after booting) by executing the sbin/init script.

Linux: initialisation:

Process creation system calls including attribute calls. Also included under this heading are initialisation and set up calls for threads and their attributes, semaphores and other resources.

Calls are:

fork	bind
sem_init	sched_setparam
sem_open	sched_setscheduler
timer_create	set gid
cfspeed family	setpgid
tcsettar	setsid
tcsetgrp	Setuid
socket	

pthread_atfork	pthread_attr_setscope
pthread_attr_init	pthread_attr_setstacksize
pthread_attr_setdetachedstate	pthread_create
pthread_mutexattr_init	pthread_setcancelstate
pthread_attr_setinheritsched	pthread_setspecific
pthread_attr_setschedparam	pthread_setcanceltype
pthread_setschedparam	pthread_setstackaddr
pthread_mutexattr_setpshared	pthread_keycreate
pthread_attr_setschedpolicy	pthread_mutex_setpriorityceiling

Linux: timing watchdog:

Clocks and timers in Linux are relatively Spartan, as described elsewhere in this report. However, the following provision exists.

clock_settime	timer_delete
clock_gettime	timer_settime
clock_getres	timer_gettime
timer_create	timer_getoverrun

Linux: partitioning*:*

There are no specific Linux calls which deal with partitions, since the concept is not recognised. Process initialisation and user/group identifier facilities may be used to support partitioning as described in the main body of the report.

Linux: intra-partition communication:

Since Linux does not recognise the partition concept there are no specific facilities to support intra-partition communication. Since all threads share one address space, intra-partition communication can simply be achieved by use of memory addresses, but to provide appropriate synchronisation between threads and avoid race conditions, thread communication facilities in the form of mutex and condition variable operations would fall into this classification.

Calls are:

pthread_mutex_lock	pthread_mutex_unlock
pthread_mutex_trylock	pthread_mutexattr_getpshared

I/O facilities may also be used for intra-partition communication but this would be unusual.

Calls are:

read	aio_read
write	aio_write

Linux: inter-partition communication:

This is a specific version of inter-process communication. A number of mechanisms are possible, as noted elsewhere.

Calls are:

signals	
alarm	sigprocmask
kill	sigqueue
pthread_kill	sigstop family
pthread_sigmask	sigsuspend
sigaction	sigwait family
sigpending	

pipes	sem_trywait
named pipes	sem_wait)
semaphores	mmap
sem_getvalue	shm_open
sem_post	

sockets (listen, accept, connect, read, write, close)

Linux: scheduling:

Scheduling includes explicit scheduling calls and thread scheduling (mutexes and condition variables) which can also be viewed as intra-partition communication facilities.

Calls are:

exec family	sleep
wait family	select

nanosleep	sched_getschedule
pause	sched_rr_get_interval
pthread_attr_getinheritedsched	sched_yield
pthread_attr_getschedparam	pthread_mutex_setpriorityceiling
pthread_getschedparam	pthread_mutex_getpriorityceiling
sched_getparam	pthread_getspecific, pthread_once
pthread_attr_getschedpolicy	pthread_self, pthread_mutexattr family
pthread_attr_getscope	pthread_attr_getdetachedpolicy
pthread_attr_getstacksize	pthread_attr_getdetached
pthread_getstackaddr	sched_get_priority_max
pthread_join	sched_get_priority_min
pthread_testcancel	mutexes

See also inter-process communications.

Linux: processing:

This category relates to the calls to the underlying processing hardware. Failure of processing implies a complete failure of the O/S.

Linux: BIT and Health monitoring:

Linux provides little in the way of these facilities.

Possible calls in this category are:

pthread_attr_getdetachedstate aio_error

Linux: Close-down:

exit	
pthread_attr_destroy	sem_destroy
pthread_key_delete	pthread_mutexattr_destroy
pthread_detach	sem_unlink
pthread_exit	munmap
sem_close	shm_unlink

Linux: Memory management and data storage:

These services comprise virtual memory management services, file (including device) and directory operations.

Calls are:

mlock	msync
mlockall	munlock
mprotect	munlockall

chdir	opendir
closedir	readdir family
getcwd	rewinddir
mkdir	rmdir

File operations

access	open
chmod family	pathconf family
chown	read
close	rename
create dup family	stat family
fcntl	umask
fsync family	unlink
ftruncate	utime
link	write
lseek	

Terminal interface

tcgetattr	tcsendbreak
tcdrain	tcflush
tcflow	tcgetgrp

Asynchronous file operations

aio_cancel	aio_fsync
aio_return	aio_suspend
aio_listio	

Access control

getgid	geteuid
getgid	getgroups
getlogin family	getpgrp
getuid	

Linux: configuration management:

There are no specific configuration management facilities provided by Linux.

Calls not placeable in the services model.

A number of calls do not fit well into the above classification. They are: ctermid, getgrgid, getgrnam, getpw family, sysconf, time, times, ttyname, uname.

APPENDIX C

LINUX KERNEL SIZE AND COMPLEXITY METRICS

The following table gives results from the C-Metrics tool when run against the Linux kernel components. The metrics do not include those for software items from the following source directories:

i) all architectures other than i386;

ii) the i386 boot directory;

iii) the i386 lib directory;

iv) the i386 maths emulation routines;

v) the drivers directory;

vi) the fs (file system) directory;

vii) the init directory (concerned with getting the kernel started);

viii) the lib directory;

ix) the net library containing network services, which have the same status as device drivers; and

x) the scripts directory.

The columns in the table are mostly self-explanatory. The columns labelled with "CC" refer to the McCabe's cyclomatic complexity metric V(G). This is the most frequently used metric for assessing software complexity, and essentially measures the number of decision points in a procedure or function. A value of 20 or less is often regarded as indicating acceptable complexity. Each branch of a "case" statement will add 1 to the cyclomatic complexity value so that very high values can be sometimes be reported for software which is in fact quite simple in structure. The reason why some modules show a high maximum cyclomatic complexity has not been investigated by the study.

The software tool used to collect these measurements ignores C pre-processor directives and assembly language inserts, so the results are indicative only.

The following table gives results from the C-Metrics tool when run against the Linux kernel components. The metrics do not include those for software items from the following source directories:

i) all architectures other than i386;

ii) the i386 boot directory;

iii) the i386 lib directory;

iv) the i386 maths emulation routines;

v) the drivers directory;

vi) the fs (file system) directory;

vii) the init directory (concerned with getting the kernel started);

viii) the lib directory;

ix) the net library containing network services, which have the same status as device drivers; and

x) the scripts directory.

The columns in the table are mostly self-explanatory. The columns labelled with "CC" refer to the McCabe's cyclomatic complexity metric V(G). This is the most frequently used metric for assessing software complexity, and essentially measures the number of decision points in a procedure or function. A value of 20 or less is often regarded as indicating acceptable complexity. Each branch of a "case" statement will add 1 to the cyclomatic complexity value so that very high values can be sometimes be reported for software which is in fact quite simple in structure. The reason why some modules show a high maximum cyclomatic complexity has not been investigated by the study.

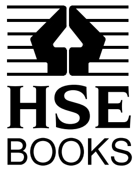
The software tool used to collect these measurements ignores C pre-processor directives and assembly language inserts, so the results are indicative only.

File name	Lines	Effect.	E%	Comment	C%	Blank	B%	Func.	avg CC	min CC	max CC	class	struct
vm86.c	677	450	66	85	12	84	12	22	4	1	17	0	1
visws_apic.c	410	186	45	122	29	64	15	12	2	1	11	0	4
traps.c	1035	604	58	224	21	136	13	38	2	1	10	0	1
time.c	706	281	39	295	41	110	15	10	4	1	10	0	1
sys_i386.c	256	165	64	29	11	36	14	9	4	1	20	0	2
smpboot.c	1024	454	44	353	34	160	15	17	5	1	26	0	0
smp.c	542	172	31	254	46	68	12	21	2	1	9	0	2
signal.c	715	463	64	81	11	111	15	14	7	1	40	0	2
setup.c	2539	1414	55	611	24	364	14	35	8	1	40	0	11
semaphore.c	243	73	30	87	35	67	27	4	2	1	4	0	0
ptrace.c	470	321	68	73	15	45	9	6	16	1	67	0	0
process.c	775	359	46	244	31	118	15	27	2	1	12	0	1
pci-visws.c	141	83	58	8	5	23	16	12	1	1	4	0	2
pci-pc.c	1089	666	61	155	14	158	14	43	3	1	16	0	8
pci-irq.c	750	413	55	146	19	83	11	32	4	1	27	0	3
pci-i386.c	384	159	41	159	41	31	8	9	5	1	12	0	0
pci-dma.c	37	19	51	8	21	5	13	2	2	1	4	0	0
mtrr.c	2277	1095	48	739	32	232	10	40	8	1	43	0	8
msr.c	273	141	51	35	12	68	24	14	2	1	5	0	2
mpparse.c	651	391	60	119	18	77	11	15	6	1	26	0	0
microcode.c	374	219	58	69	18	52	13	8	5	1	16	0	3
mca.c	980	371	37	332	33	211	21	22	5	1	12	0	3
ldt.c	148	102	68	18	12	17	11	3	9	4	19	0	0
irq.c	1183	556	46	385	32	154	13	32	4	1	12	0	1
ioport.c	116	58	50	35	30	12	10	3	5	4	7	0	0
io_apic.c	1623	848	52	380	23	234	14	50	4	1	29	0	4
init_task.c	34	15	44	14	41	5	14	0	0	0	0	0	1
i8259.c	506	239	47	186	36	73	14	12	2	1	5	0	3
i387.c	522	307	58	54	10	70	13	35	2	1	10	0	0
i386_ksyms.c	167	137	82	5	2	25	14	0	0	0	0	0	0
dmi_scan.c	467	256	54	89	19	55	11	15	3	1	10	0	4
cpuid.c	165	88	53	32	19	29	17	8	2	1	4	0	2
bluesmoke.c	241	122	50	46	19	42	17	9	3	1	10	0	0
apm.c	1791	1033	57	424	23	173	9	42	6	1	30	0	5
apic.c	792	312	39	318	40	114	14	20	3	1	11	0	0

GLOSSARY

ATC	Air Traffic Control
API	Application Programming Interface
BIT	Built In Test
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CSE	CSE International Ltd
DRACAS	Defect Reporting, Analysis and Corrective Action System
EMC	Electromagnetic Compatibility
EMI	Electromagnetic Interference
FFA	Functional Failure Analysis
GID	Group Identifier
GNU	Symbol of the Free Software Foundation
HRT	Hard Real Time
HSE	UK Health and Safety Executive
ICMP	Internet Control Message Protocol
IMA	Integrated Modular Avionics
I/O	Input/Output
IP	Internet Protocol
IPC	Inter-Process Communication
ISP	Internet Service Provider
LDP	Linux Documentation Project
LTP	Linux Test Project
LSB	Linux Standard Base
MoD	United Kingdom Ministry of Defence
MTBF	Mean Time Between Failures
PC	Personal Computer
RAM	Random Access Memory
SCADA	System Control and Data Acquisition
SIL	Safety Integrity Level
SMART	Self-Monitoring Analysis and Reporting Technology
SNMP	Simple Network Management Protocol
SRG	Safety Regulation Group (UK Civil Aviation Authority)
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UID	User Identifier

Printed and published by the Health and Safety Executive
C1.25 8/02



MAIL ORDER
HSE priced and free
publications are
available from:
HSE Books
PO Box 1999
Sudbury
Suffolk CO10 2WA
Tel: 01787 881165
Fax: 01787 313995
Website: www.hsebooks.co.uk

RETAIL
HSE priced publications
are available from booksellers

HEALTH AND SAFETY INFORMATION
HSE InfoLine
Tel: 08701 545500
Fax: 02920 859260
e-mail: hseinformationservices@natbrit.com
or write to:
HSE Information Services
Caerphilly Business Park
Caerphilly CF83 3GG

HSE website: www.hse.gov.uk

RR 011

£15.00

ISBN 0-7176-2538-9