SUSE Linux Enterprise Server 10 SP1 EAL4
High-Level Design
Version 1.2.1

Version	Author	Date	Comments
1.0	EJR	3/15/07	First draft based on RHEL5 HLD
1.1	EJR	4/19/07	Updates based on comments from Stephan Mueller and Klaus Weidner
1.2	GCW	4/26/07	Incorporated Stephan's comment to remove racoon
1.2.1	GCW	10/27/08	Added legal matter missing from final draft.

Table of Contents

1 Introduction

This document describes the High Level Design (HLD) for the SUSE® Linux® Enterprise Server 10 Service Pack 1 operating system. For ease of reading, this document uses the phrase SUSE Linux Enterprise Server and the abbreviation SLES as a synonym for SUSE Linux Enterprise Server 10 SP1.

This document summarizes the design and Target of Evaluation Security Functions (TSF) of the SUSE Linux Enterprise Server (SLES) operating system. Used within the Common Criteria evaluation of SUSE Linux Enterprise Server at Evaluation Assurance Level (EAL) 4, it describes the security functions defined in the Common Criteria Security Target document.

1.1 Purpose of this document

The SLES distribution is designed to provide a secure and reliable operating system for a variety of purposes. This document describes the high-level design of the product and provides references to other, more detailed design documentation that describe the structure and functions of the system. This document is consistent with additional high-level design documents, as well as with the supporting detailed design documents for the system. There are pointers to those documents in this document.

The SLES HLD is intended as a source of information about the architecture of the system for any evaluation team.

1.2 Document overview

This HLD contains the following chapters:

Chapter 2 presents an overview of the IBM® eServer™ systems, including product history, system architecture, and TSF identification.

Chapter 3 summarizes the eServer hardware subsystems, characterizes the subsystems with respect to security relevance, and provides pointers to detailed hardware design documentation.

Chapter 4 expands on the design of the TSF software subsystems, particularly the kernel, which is identified in Chapter 2.

Chapter 5 addresses functional topics and describes the functionality of individual subsystems, such as memory management and process management.

Chapter 6 maps the Target of Evaluation (TOE) summary specification from the SUSE Linux Enterprise Server Security Target to specific sections in this document.

1.3 Conventions used in this document

The following font conventions are used in this document:

`Constant Width` (`Monospace`) shows code or output from commands, and indicates source-code keywords that appear in the code as well as file and directory names, program and command names, command-line options.

Italic indicates URLs, book titles, and introduces new terms.

1.4 Terminology

For definitions of technical terms and phrases that have specific meaning for Common Criteria evaluation, please refer to the Security Target.

2 System Overview

The Target of Evaluation (TOE) is SUSE Linux Enterprise Server (SLES) running on an IBM eServer host computer. The SLES product is available on a wide range of hardware platforms. This evaluation covers the SLES product on the IBM eServer System x™, System p™, and System z™, and eServer 326 (Opteron). (Throughout this document, SLES refers only to the specific evaluation platforms).

Multiple TOE systems can be connected via a physically-protected Local Area Network (LAN). The IBM eServer line consists of Intel processor-based System x systems, POWER5™ and POWER5+™ processor-based System p systems, IBM mainframe System z systems, and AMD Opteron processor-based systems that are intended for use as networked workstations and servers.

Figure 2-1 shows a series of interconnected TOE systems. Each TOE system is running the SLES operating system on an eServer computer. Each computer provides the same set of local services, such as file, memory, and process management. Each computer also provides network services, such as remote secure shells and file transfers, to users on other computers. A user logs in to a host computer and requests services from the local host and also from other computers within the LAN.

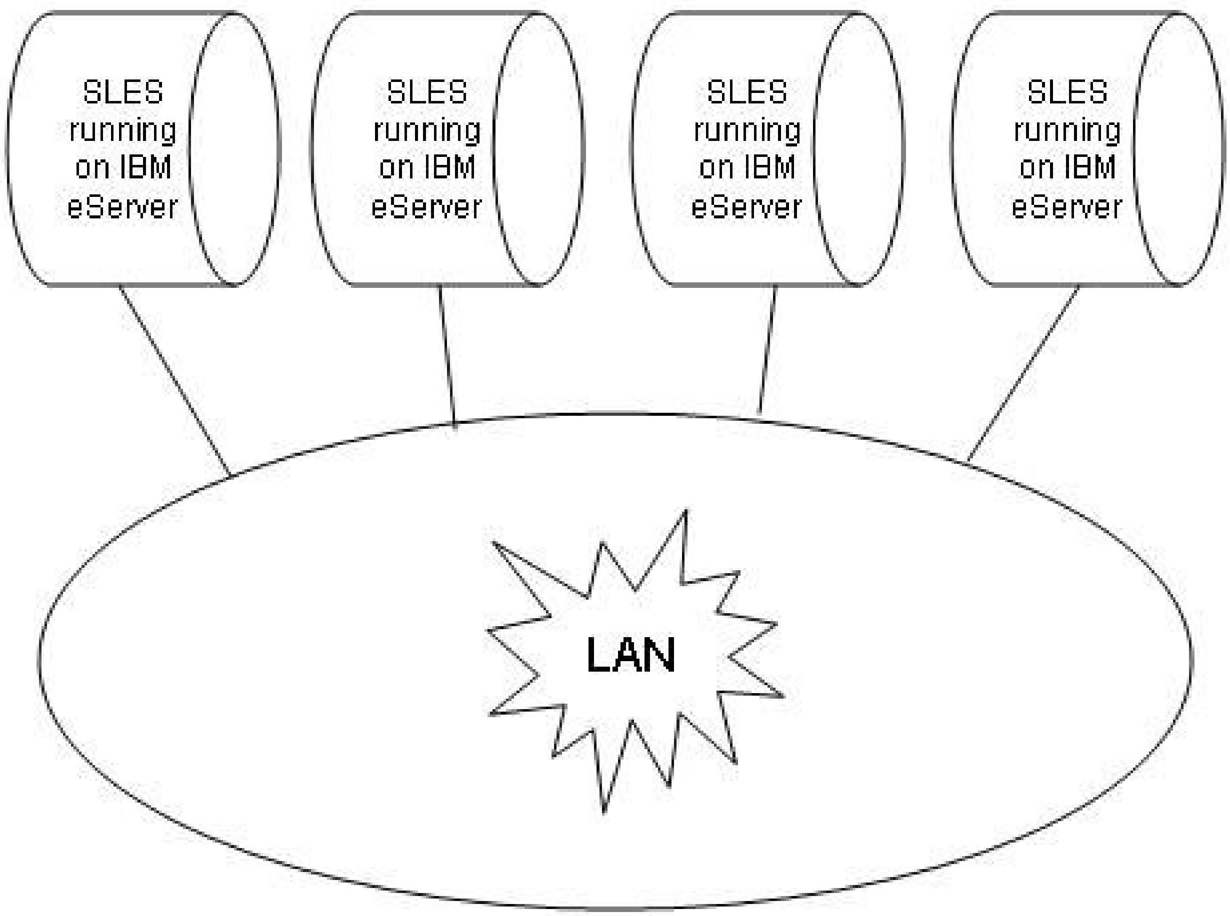


Figure 2-1: Series of TOE systems connected by a physically protected LAN

User programs issue network requests by sending Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) messages to another computer. Some network protocols, such as Secure Shell (ssh), can start a shell process for the user on another computer, while others are handled by trusted server daemon processes.

The TOE system provides user Identification and Authentication (I&A) mechanism by requiring each user to log in with proper password at the local workstation, and also at any remote computer where the user can enter commands to a shell program (for example, remote ssh sessions). Each computer enforces a coherent Discretionary Access Control (DAC) policy, based on UNIX®-style mode bits and an optional Access Control List (ACL) for the named objects under its control.

This chapter documents the SUSE Linux Enterprise Server and IBM eServer product histories, provides an overview of the TOE system, and identifies the portion of the system that constitutes the TOE Security Functions (TSF).

2.1 Product history

This section gives a brief history of the SLES and the IBM eServer series systems.

2.1.1 SUSE Linux Enterprise Server

SUSE Linux Enterprise Server is based on version 2.6 of the Linux kernel. Linux is a UNIX-like open-source operating system originally created in 1991 by Linus Torvalds of Helsinki, Finland. SUSE was founded in 1992 by four German software engineers, and is the oldest major Linux solutions provider.

2.1.2 eServer systems

IBM eServer systems were introduced in 2000. The IBM eServer product line brings technological innovation, application flexibility, and autonomic capabilities for managing the heterogeneous mix of servers required to support dynamic on-demand business. It enables customers to meet their business needs by providing unlimited scalability, support for open standards, and mission-critical qualities of service.

Following are systems in the IBM eServer product line that are included in the TOE:

- System z: Mainframe-class servers running mission-critical applications.
- System p: UNIX servers, technologically advanced POWER5 and POWER5+ processor-based servers for commercial and technical computing applications.
- System x: Intel-based servers with high performance and outstanding availability.
- eServer 326: AMD Opteron-based servers with outstanding value in high performance computing in both 32-bit and 64-bit environments.
- BladeCenter®: Intel Xeon, AMD Opteron, PowerPC, POWER5, and POWER5+ processor based servers.

Since introducing eServers in 2000, new models with more powerful processors have been added to the System x, System p, and System z lines. The AMD Opteron processor-based eServer 325 was added to the eServer series in 2003; the eServer 326 is the next iteration of that model with updated components. The AMD Opteron eServer 326 is designed for powerful scientific and technical computing. The Opteron processor supports both 32-bit and 64-bit architectures, thus allowing easy migration to 64-bit computing.

2.2 High-level product overview

The TOE consists of SLES running on an eServer computer. The TOE system can be connected to other systems by a protected LAN. SLES provides a multi-user, multi-processing environment, where users interact with the operating system by issuing commands to a command interpreter, by running system utilities, or by the users developing their own software to run in their own protected environments.

The Common Criteria for Information Technology Security Evaluation [CC] and the Common Methodology for Information Technology Security Evaluation [CEM] demand breaking the TOE into logical subsystems that can be either (a) products, or (b) logical functions performed by the system.

The approach in this section is to break the system into structural hardware and software subsystems that include, for example, pieces of hardware such as planars and adapters, or collections of one or more software processes such as the base kernel and kernel modules. Chapter 4 explains the structure of the system in terms of these architectural subsystems. Although the hardware is also described in this document, the reader should be aware that while the hardware itself is part of the TOE environment, it is not part of the TOE.

The following subsections present a structural overview of the hardware and software that make up an individual eServer host computer. This single-computer architecture is one of the configurations permitted under this evaluation.

2.2.1 eServer host computer structure

This section describes the structure of SLES for an individual eServer host computer. As shown in Figure 2-2, the system consists of eServer hardware, the SLES kernel, trusted non-kernel processes, TSF databases, and untrusted processes. In this figure, the TOE itself consists of Kernel Mode software, User Mode software, and hardware. The TOE Security Functions (TSF) are shaded in gray. Details such as interactions within the kernel, inter-process communications, and direct user access to the hardware are omitted.

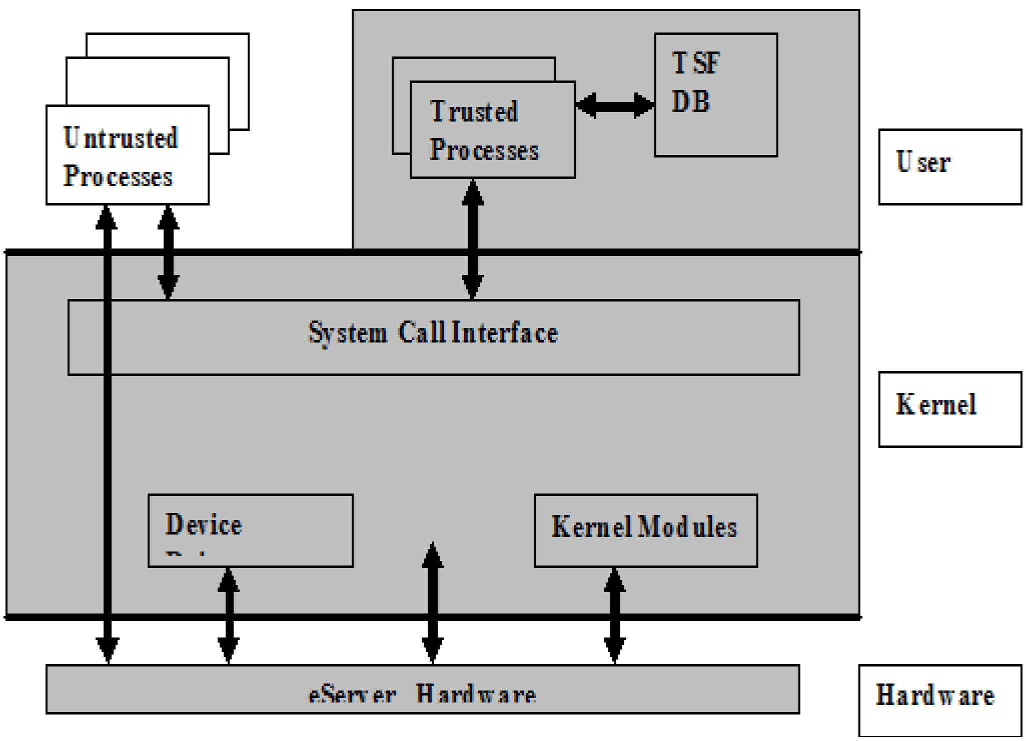


Figure 2-2: Overall structure of the TOE

The planar components, including CPUs, memory, buses, on board adapters, and support circuitry; additional adapters, including LAN and video; and, other peripherals, including storage devices, monitors, keyboards, and front-panel hardware, constitute the hardware.

The SLES kernel includes the base kernel and separately-loadable kernel modules and device drivers. (Note that a device driver can also be a kernel module.) The kernel consists of the bootable kernel image and its loadable modules. The kernel implements the system call interface, which provides system calls for file management, memory management, process management, networking, and other TSF (logical subsystems) functions addressed in the Functional Descriptions chapter of this document. The structure of the SLES kernel is described further in the Software Architecture chapter of this paper.

Non-kernel TSF software includes programs that run with the administrative privilege, such as the `sshd`, `cron`, `atd`, and `vsftpd` daemons. The TSF also includes the configuration files that define authorized users, groups of users, services provided by the system, and other configuration data. Not included as TSF are shells used by administrators, and standard utilities invoked by administrators.

The SLES system, which includes hardware, kernel-mode software, non-kernel programs, and databases, provides a protected environment in which users and administrators run the programs, or sequences of CPU instructions. Programs execute as processes with the identity of the users that started them (except for some exceptions defined in this paper), and with privileges as dictated by the system security policy. Programs are subject to the access control and accountability processes of the system.

2.2.2 eServer system structure

The system is an eServer computer, which permits one user at a time to log in to the computer console. Several virtual consoles can be mapped to a single physical console. Different users can login through different virtual consoles simultaneously. The system can be connected to other computers via physically and logically protected LANs. The eServer hardware and the physical LAN connecting the different systems running SLES are not included within the evaluation boundary of this paper. External routers, bridges, and repeaters are also not included in the evaluation boundary of this paper.

A standalone host configuration operates as a CC-evaluated system, which can be used by multiple users at a time. Users can operate by logging in at the virtual consoles or serial terminals of a system, or by setting-up background execution jobs. Users can request local services, such as file, memory, and process management, by making system calls to the kernel. Even though interconnection of different systems running SLES is not included in the evaluation boundary, the networking software is loaded. This aids in a user's request for network services (for example, FTP) from server processes on the same host.

Another configuration provides a useful network configuration, in which a user can log in to the console of any of the eServer host computers, request local services at that computer, and also request network services from any of the other computers. For example, a user can use `ssh` to log into one host from another, or `scp` to transfer files from one host to another. The configuration extends the single LAN architecture to show that SLES provides Internet Protocol (IP) routing from one LAN segment to another. For example, a user can log in at the console of a host in one network segment and establish an ssh connection to a host in another network segment. Packets on the connection travel across a LAN segment, and they are routed by a host in that segment to a host on another LAN segment. The packets are eventually routed by the host in the second LAN segment to a host on a third LAN segment, and from there are routed to the target host. The number of hops from the client to the server are irrelevant to the security provided by the system, and are transparent to the user.

The hosts that perform routing functions have statically-configured routing tables. When the hosts use other components for routing (for example, a commercial router or switches), those components are assumed to perform the routing functions correctly, and do not alter the data part of the packets.

If other systems are to be connected to the network, with multiple TOE systems connected via a physically protected LAN, then they need to be configured and managed by the same authority using an appropriate security policy not conflicting with the security policy of the TOE.

2.2.3 TOE services

Each host computer in the system is capable of providing the following types of services:

- Local services to the users who are currently logged in to the system using a local computer console, virtual consoles, or terminal devices connected through physically protected serial lines.
- Local services to the previous users via deferred jobs; an example is the `cron` daemon.
- Local services to users who have accessed the local host via the network using a protocol such as `ssh`, which starts a user shell on the local host.
- Network services to potentially multiple users on either the local host or on remote hosts.

Figure 2-3 illustrates the difference between local services that take place on each local host computer, versus network services that involve client-server architecture and a network service layer protocol. For example, a user can log in to the local host computer and make file system requests or memory management requests for services via system calls to the kernel of the local host. All such local services take place solely on the local host computer and are mediated solely by trusted software on that host.

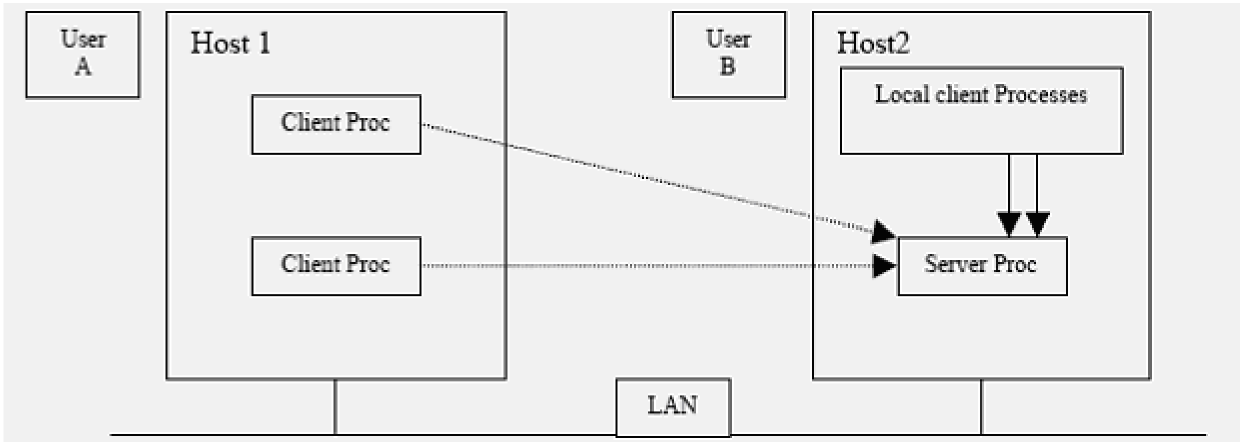


Figure 2-3: Local and network services provided by SLES

Network services, such as `ssh` or `ftp`, involve client-server architecture and a network service-layer protocol. The client-server model splits the software that provides a service into a client portion that makes the request, and a server portion that carries out the request, usually on a different computer. The service protocol is the interface between the client and server. For example, User A can log in at Host 1, and then use `ssh` to log in to Host 2. On Host 2, User A is logged in from a remote host.

On Host 1, when User A uses `ssh` to log in to Host 2, the ssh client on Host 1 makes protocol requests to an ssh server process on Host 2. The server process mediates the request on behalf of User A, carries out the requested service, if possible, and returns the results to the requesting client process.

Also, note that the network client and server can be on the same host system. For example, when User B uses `ssh` to log in to Host 2, the user's client process opens an ssh connection to the ssh server process on Host 2. Although this process takes place on the local host computer, it is distinguished from local services because it involves networking protocols.

2.2.4 Security policy

A user is an authorized individual with an account. Users can use the system in one of three ways:

1. By interacting directly with the system thorough a session at a computer console (in which case the user can use the graphical display provided as the console), or
2. By interacting directly with system through a session at a serial terminal, or
3. Through deferred execution of jobs using the `cron` and `atd` utilities.

A user must log in at the local system in order to access the protected resources of the system. Once a user is authenticated, the user can access files or execute programs on the local computer, or make network requests to other computers in the system.

The only subjects in the system are processes. A process consists of an address space with an execution context. The process is confined to a computer; there is no mechanism for dispatching a process to run remotely (across TCP/IP) on another host. Every process has a process ID (PID) that is unique on its local host computer, but PIDs are not unique throughout the system. As an example, each host in the system has an init process with PID 1. Section 5.2 of this document explains how a parent process creates a child by making a `clone()`, `fork()` or a `vfork()` system call; the child can then call `execve()` to load a new program.

Objects are passive repositories of data. The TOE defines three types of objects: named objects, storage objects, and public objects. Named objects are resources, such as files and IPC objects, which can be manipulated by multiple users using a naming convention defined at the TSF interface. A storage object is an object that supports both read and write access by multiple non-trusted subjects. Consistent with these definitions, all named objects are also categorized as storage objects, but not all storage objects are named objects. A public object is an object that can be publicly read by non-trusted subjects and can be written only by trusted subjects.

SLES enforces a DAC policy for all named objects under its control, and an object reuse policy for all storage objects under its control. Additional access control checks are possible, if an optional kernel module is loaded, such as AppArmor. If AppArmor is loaded, DAC policy is enforced first, and the additional access control checks are made only if DAC would allow the access. The additional checks are non-authoritative; that is, a DAC policy denial cannot be overridden by the additional access control checks in the kernel module.

While the DAC policy that is enforced varies among different object classes, in all cases it is based on user identity and on group membership associated with the user identity. To allow for enforcement of the DAC policy, all users must be identified, and their identities must be authenticated. The TOE uses both hardware and software protection mechanisms.

The hardware mechanisms used by SLES to provide a protected domain for its own execution include a multistate processor, memory segment protection, and memory page protection. The TOE software relies on these hardware mechanisms to implement TSF isolation, non-circumventability, and process address-space separation.

A user can log in at the console, at other directly attached terminals, or through a network connection. Authentication is based on a password entered by the user and authentication data stored in a protected file. Users must log in to a host before they can access any named objects on that host. Some services, such as `ssh` to obtain a shell prompt on another host, or `ftp` to transfer files between hosts in the distributed system, require the user to re-enter authentication data to the remote host. SLES permits the user to change passwords (subject to TOE enforced password guidelines), change identity, submit batch jobs for deferred execution, and log out of the system. The Strength of Function Analysis [VA] shows that the probability of guessing a password is sufficiently low given the minimum password length and maximum password lifetime.

The system architecture provides TSF self-protection and process isolation mechanisms.

2.2.5 Operation and administration

The eServer networks can be composed of one, several, or many different host computers, each of which can be in various states of operation, such as being shut down, initializing, being in single-user mode, or online in a secure state. Thus, administration involves the configuration of multiple computers and the interactions of those computers, as well as the administration of users, groups, files, printers, and other resources for each eServer system.

The TOE provides the `useradd`, `usermod`, and `userdel` commands to add, modify, and delete a user account. It provides the `groupadd`, `groupmod`, and `groupdel` commands to add, modify, and delete a group from the system. These commands accept options to set up or modify various parameters for accounts and groups. The commands modify the appropriate TSF databases and provide a safer way than manual editing to update authentication databases. Refer to the appropriate command man pages for detailed information about how to set up and maintain users and groups.

2.2.6 TSF interfaces

The TSF interfaces include local interfaces provided by each host computer, and the network client-server interfaces provided by pairs of host computers.

The local TSF interfaces provided by an individual host computer include:

- Files that are part of the TSF database that define the configuration parameters used by the security functions.
- System calls made by trusted and untrusted programs to the privileged kernel-mode software. As described separately in this document, system calls are exported by the base SLES kernel and by kernel modules.
- Interfaces to trusted processes and trusted programs
- Interfaces to the SLES kernel through the `/proc` and the `/sys` pseudo file systems

External TSF interfaces provided by pairs of host computer include SSH v2 and SSL v3.

For more detailed information about these interfaces, refer to:

- SSH v2 Proposed Standard RFC 4819 Secure Shell Public Key Subsystem http://www.ietf.org/rfc/rfc4819.txt
- SSLv3 Draft http://wp.netscape.com/eng/ssl3/draft302.txt
- RFC 3268 Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS) http://www.ietf.org/rfc/rfc3268.txt

The following are interfaces that are not viewed as TSF interfaces:

- Interfaces between non-TSF processes and the underlying hardware. Typically, user processes do not interface directly with the hardware; exceptions are processor and graphics hardware. User processes interact with the processor by executing CPU instructions, reading and modifying CPU registers, and modifying the contents of physical memory assigned to the process. User processes interact with graphics hardware by modifying the contents of registers and memory on the graphics adapter. Unprivileged processor instructions are externally visible interfaces. However, the unprivileged processor instructions do not implement any security functionality, and the processor restricts these instructions to the bounds defined by the processor. Therefore, this interface is not considered as part of the TSF.
- Interfaces between different parts of the TSF that are invisible to normal users (for example, between subroutines within the kernel) are not considered to be TSF interfaces. This is because the interface is internal to the trusted part of the TOE and cannot be invoked outside of those parts. Those interfaces are therefore not part of the functional specification, but are explained in this HLD.
- The firmware (PR/SM™, z/VM™, P5-LPAR), while part of the TOE, are not considered as providing TSF interfaces because they do not allow direct unprivileged operations to them.
- System z processor exceptions reflected to the firmware, including z/VM, PR/SM, and LPAR, are not considered to be TSF interfaces. They are not relevant to security because they provide access to the z/VM kernel, which does not implement any security functionality.
- The System z z/VM DIAGNOSE code interface is not considered a TSF interface because it is not accessible by unprivileged processes in the problem state, and does not provide any security functionality.

TSF interfaces include any interface that is possible between untrusted software and the TSF.

2.3 Approach to TSF identification

This section summarizes the approach to identification of the TSF.

As stated in Section 2.2.6, while the hardware and firmware (z/VM, PR/SM, LPAR) are part of the TOE, they are not considered as providing TSF interfaces. The SLES operating system, on the other hand, does provide TSF interfaces.

The SLES operating system is distributed as a collection of packages. A package can include programs, configuration data, and documentation for the package. Analysis is performed at the file level, except where a particular package can be treated collectively. A file is included in the TSF for one or more of the following reasons:

- It contains code, such as the kernel, kernel module, and device drivers, that runs in a privileged hardware state.
- It enforces the security policy of the system.
- It allows setuid or setgid to a privileged user (for example, root) or group.
- It started as a privileged daemon; an example is one started by `/etc/init.d`.
- It is software that must function correctly to support the system security mechanisms.
- It is required for system administration.
- It consists of TSF data or configuration files.
- It consists of libraries linked to TSF programs.

There is a distinction between non-TSF user-mode software that can be loaded and run on the system, and software that must be excluded from the system. The following methods are used to ensure that excluded software cannot be used to violate the security policies of the system:

- The installation software will not install any device drivers except those required for the installed hardware. Consequently, excluded device drivers will not be installed even if they are on the installation media.
- The installation software may change the configuration (for example, mode bits) so that a program cannot violate the security policy.

3 Hardware architecture

The TOE includes the IBM System x, System p, System z, and eServer 326. This section describes the hardware architecture of these eServer systems. For more detailed information about Linux support and resources for the entire eServer line, refer to http://www.ibm.com/systems/browse/linux.

3.1 System x

IBM System x systems are Intel processor-based servers with X-architecture technology enhancements for reliability, performance, and manageability. X-architecture is based on technologies derived from the IBM ES™-, RS™-, and AS™-series servers.

3.1.1 System x hardware overview

The IBM System x servers offer a range of systems, from entry-level to enterprise class. The high-end systems offer support for gigabytes of memory, large RAID configurations of SCSI and fiber channel disks, and options for high-speed networking. IBM System x servers are equipped with a real-time hardware clock. The clock is powered by a small battery and continues to tick even when the system is switched off. The real-time clock maintains reliable time for the system. For the specification of each of the System x servers, refer to the system x hardware Web site at http://www.ibm.com/systems/x/.

3.1.2 System x hardware architecture

The IBM System x servers are powered by Intel Xeon® and Xeon MP processors. For detailed specification information for each of these processors, refer to the Intel processor spec-finder Web site at http://processorfinder.intel.com/scripts/default.asp.

The Intel Xeon processor is mainly based on EM64 technology, which has the following three operating modes:

- 32-bit legacy mode: In this mode, both AMD64 and EM64T processors will act just like any other IA32 compatible processor. One can install this 32-bit operating system and run 32-bit applications on such a system, but it fails to make use of new features such as flat memory addressing above 4 GB or the additional general Purpose Registers (GPRs). 32-bit applications will run just as fast as they would on any current 32-bit processor.
- Compatibility mode: This is an intermediate mode of the full 64-bit mode described next. In this mode one has to install a 64-bit operating system and 64-bit drivers. If a 64-bit operating system and drivers are installed, Xeon processors will be enabled to support a 64-bit operating system with both 32-bit applications or 64-bit applications. Hence this mode has the ability to run a 64-bit operating system while still being able to run unmodified 32-bit applications. Each 32-bit application will still be limited to a maximum of 4 GB of physical memory. However, the 4 GB limit is now imposed on a per-process level, not at a system-wide level.
- Full 64-bit mode: This mode is referred as IA-32e mode. This mode is operative when a 64-bit operating system and 64-bit application are used. In the full 64-bit operating mode, an application can have a virtual address space of up to 40 bits, which equates to 1 TB of addressable memory. The amount of physical memory will be determined by how many Dual In-line Memory Module (DIMM) slots the server has, and the maximum DIMM capacity supported and available at the time.

In this mode, applications may access:

- 64-bit flat linear addressing
- 8 new general-purpose registers (GPRs)
- 8 new registers for streaming Single Instruction/Multiple Data (SIMD) extensions (SSE, SSE2 and SSE3)
- 64-bit-wide GPRs and instruction pointers
- uniform byte-register addressing
- fast interrupt-prioritization mechanism
- a new instruction-pointer relative-addressing mode.

For architectural details about all System x models, and for detailed information about individual components such as memory, cache, and chipset, refer to the "Accessories & Upgrades" section at http://www.ibm.com/systems/x/

USB (except keyboard and mouse), PCMCIA, and IEEE 1394 (Firewire) devices are not supported in the evaluated configuration.

3.2 System p

The IBM System p systems are PowerPC, POWER5 and POWER5+ processor-based systems that provide high availability, scalability, and powerful 64-bit computing performance.

For more detailed information about the System p hardware, refer to the System p hardware website at http://www.ibm.com/systems/p/.

3.2.1 System p hardware overview

The IBM System p servers offer a range of systems, from entry level to enterprise class. The high-end systems offer support for gigabytes of memory, large RAID configurations of SCSI and fiber channel disks, and options for high-speed networking. The IBM System p servers are equipped with a real-time hardware clock. The clock is powered by a small battery, and continues to tick even when the system is switched off. The real-time clock maintains reliable time for the system. For the specification of each of the System p servers, refer to the corresponding data sheets on the System p literature website: http://www.ibm.com/systems/p/library/index_lit.html.

For a detailed look at various peripherals such as storage devices, communications interfaces, storage interfaces, and display devices supported on these System p models, refer to the Linux on POWER website.

http://www.ibm.com/systems/linux/power/.

3.2.2 System p hardware architecture

The IBM System p servers are powered by PowerPC™, POWER5™ and POWER5+™ processors. For detailed specification information for each of these processors, refer to the PowerPC processor documentation at http://www.ibm.com/chips/power/powerpc/ and POWER documentation at http://www.ibm.com/chips/power/aboutpower/.

For architectural details about all System p models, and for detailed information about individual components such as memory, cache, and chipset, refer to the IBM System p technical documentation at

http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base/hardware.htm or http://www.ibm.com/servers/eserver/pseries/library/.

USB (except keyboard and mouse), PCMCIA, and IEEE 1394 (Firewire) devices are not supported in the evaluated configuration.

3.3 System z

The IBM System z is designed and optimized for high-performance data and transaction serving requirements. On a System z system, Linux can run on native hardware, in a logical partition, or as a guest of the z/VM® operating system. SLES runs on System z as a guest of the z/VM Operating System.

For more detailed information about the System z hardware, refer to the System z hardware website at http://www.ibm.com/systems/z/.

3.3.1 System z hardware overview

The System z hardware runs z/Architecture™ and the S/390™ Enterprise Server Architecture (ESA) software. The IBM System z server is equipped with a real-time hardware clock. The clock is powered by a small battery, and continues to tick even when the system is switched off. The real-time clock maintains reliable time for the system. For a more detailed overview of the System z hardware models, or detailed information about specific models, refer to the http://www.ibm.com/systems/z/hardware/ site.

3.3.2 System z hardware architecture

The System z servers are powered by IBM's multi-chip module (MCM), which contains up to 20 processing units (PUs). These processing units contain the z/Architecture logic. There are three modes in which Linux can be run on a System z server: native hardware mode, logical partition mode, and z/VM guest mode. The following paragraphs describe these modes.

- Native hardware mode: In native hardware mode, Linux can run on the entire machine without any other operating system. Linux controls all I/O devices and needs support for their corresponding device drivers.
- Logical partition mode: A System z system can be logically partitioned into a maximum of 30 separate Logical Partitions (LPARs). A single System z server can then host the z/OS operating system in one partition, and Linux in another. Devices can be dedicated to a particular logical partition, or they can be shared among several logical partitions. The Linux operating system controls devices allocated to its partition, and thus needs support for their corresponding device drivers.
- z/VM guest mode: Linux can run in a virtual machine using the z/VM operating system as a hypervisor. The hypervisor provides virtualization of CPU processors, I/O subsystems, and memory. In this mode, hundreds of Linux instances can run on a single System z system. SLES runs on System z in the z/VM guest mode. Virtualization of devices in the z/VM guest mode allows SLES to operate with generic devices. The z/VM maps these generic devices to actual devices.

Figure 3-1 from the *Linux Handbook* [LH] illustrates z/VM concepts:

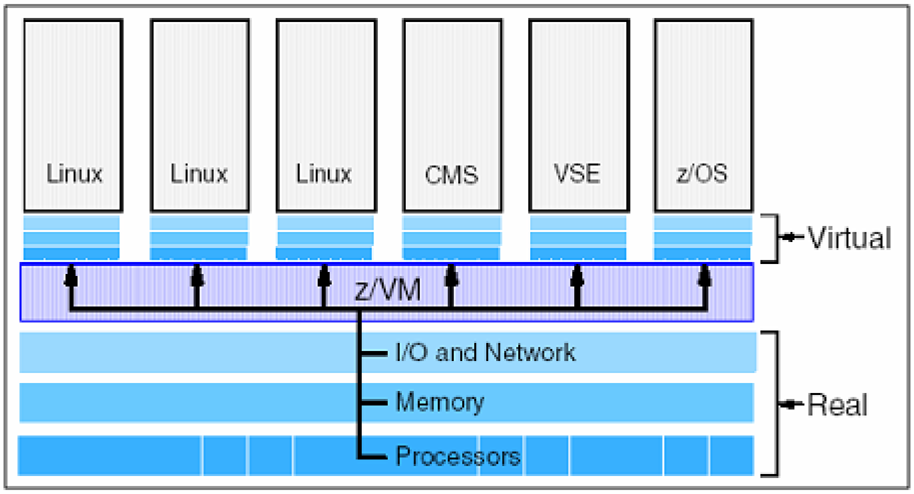


Figure 3-1: z/VM as hypervisor

For more details about z/Architecture, refer to the z/Architecture document z/Architecture Principles of Operation at http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf.

USB (except keyboard and mouse), PCMCIA, and IEEE 1394 (Firewire) devices are not supported in the evaluated configuration.

3.4 eServer 326

The IBM eServer 326 systems are AMD Opteron processor-based systems that provide high performance computing in both 32-bit and 64-bit environments. The eServer 326 significantly improves on existing 32-bit applications, and excels at 64-bit computing in performance, allowing for easy migration to 64-bit computing.

For more detailed information about eServer 326 hardware, refer to the eServer 326 hardware Web site at http://www.ibm.com/servers/eserver/opteron/.

3.4.1 eServer 326 hardware overview

The IBM eServer 326 systems offer support for up to two AMD Opteron processors, up to twelve GB of memory, hot-swap SCSI or IDE disk drives, RAID-1 mirroring, and options for high-speed networking. The IBM eServer 326 server is equipped with a real-time hardware clock. The clock is powered by a small battery and continues to tick even when the system is switched off. The real-time clock maintains reliable time for the system.

3.4.2 eServer 326 hardware architecture

The IBM eServer 326 systems are powered by AMD Opteron processors. For detailed specifications of the Opteron processor, refer to the processor documentation at http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_9003,00.html.

The Opteron is based on the AMD x86-64 architecture. The AMD x86-64 architecture is an extension of the x86 architecture, extending full support for 16-bit, 32-bit, and 64-bit applications running concurrently.

The x86-64 architecture adds a mode called the long mode. The long mode is activated by a global control bit called Long Mode Active (LMA). When LMA is zero, the processor operates as a standard x86 processor and is compatible with the existing 32-bit SLES operating system and applications. When LMA is one, 64-bit processor extensions are activated, allowing the processor to operate in one of two sub-modes of LMA. These are the 64-bit mode and the compatibility mode.

- 64-bit mode: In 64-bit mode, the processor supports 64-bit virtual addresses, a 64-bit instruction pointer, 64-bit general-purpose registers, and eight additional general-purpose registers, for a total of 16 general-purpose registers.
- Compatibility mode: Compatibility mode allows the operating system to implement binary compatibility with existing 32-bit x86 applications. These legacy applications can run without recompilation. This coexistence of 32-bit legacy applications and 64-bit applications is implemented with a compatibility thunk layer.

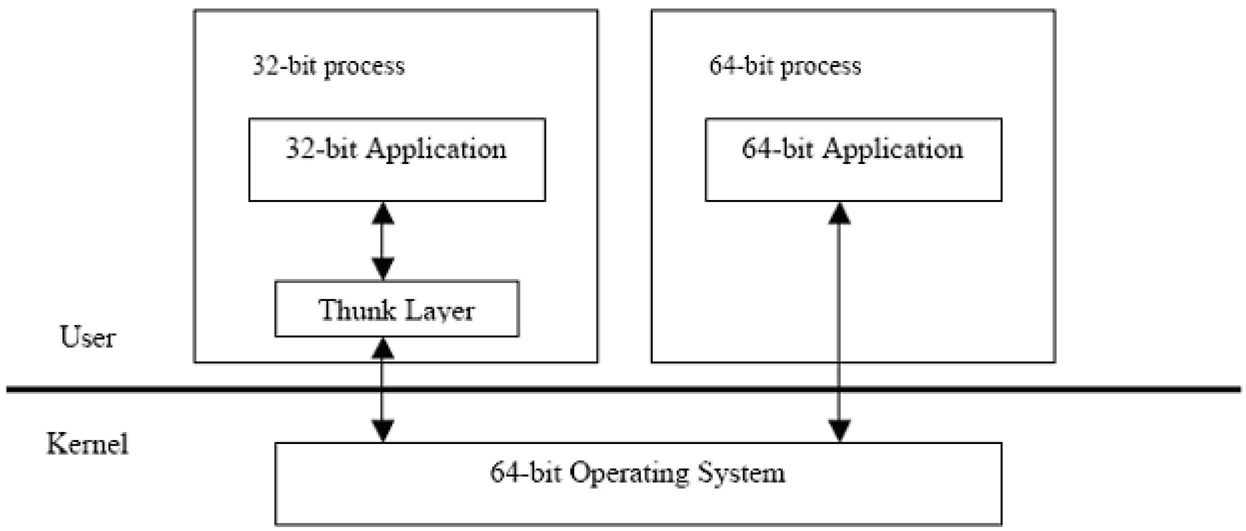


Figure 3-2: AMD x86-64 architecture in compatibility mode

The thunk layer is a library provided by the operating system. The library resides in a 32-bit process created by the 64-bit operating system to run 32-bit applications. A 32-bit application, transparent to the user, is dynamically linked to the thunk layer and implements 32-bit system calls. The thunk layer translates system call parameters, calls the 64-bit kernel, and translates results returned by the kernel appropriately and transparently for a 32-bit application.

For detailed information about the x86-64 architecture, refer to the AMD Opteron technical documentation at http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_7044,00.html.

USB (except keyboard and mouse), PCMCIA, and IEEE 1394 (Firewire) devices are not supported in the evaluated configuration.

4 Software architecture

This chapter summarizes the software structure and design of the SLES system and provides references to detailed design documentation.

The following subsections describe the TOE Security Functions (TSF) software and the TSF databases for the SLES system. The descriptions are organized according to the structure of the system and describe the SLES kernel that controls access to shared resources from trusted (administrator) and untrusted (user) processes. This chapter provides a detailed look at the architectural pieces, or subsystems, that make up the kernel and the non-kernel TSF. This chapter also summarizes the databases that are used by the TSF.

The Functional Description chapter that follows this chapter describes the functions performed by the SLES logical subsystems. These logical subsystems generally correspond to the architectural subsystems described in this chapter. The two topics were separated into different chapters in order to emphasize that the material in the Functional Descriptions chapter describes how the system performs certain key security-relevant functions. The material in this chapter provides the foundation information for the descriptions in the Functional Description chapter.

4.1 Hardware and software privilege

This section describes the terms hardware privilege and software privilege as they relate to the SLES operating system. These two types of privileges are critical for the SLES system to provide TSF self-protection. This section does not enumerate the privileged and unprivileged programs. Rather, the TSF Software Structure identifies the privileged software as part of the description of the structure of the system.

4.1.1 Hardware privilege

The eServer systems are powered by different types of processors. Each of these processors provides a notion of user mode execution and supervisor, or kernel, mode execution. The following briefly describes how these user- and kernel-execution modes are provided by the System x, System p, System z, and eServer 326 systems.

4.1.1.1 Privilege level

This section describes the concept of privilege levels by using Intel-based processors as an example. The concept of privilege is implemented by assigning a value of 0 to 3 to key objects recognized by the processor. This value is called the privilege level. The following processor-recognized objects contain privilege levels:

- Descriptors contain a field called the descriptor privilege level (DPL).
- Selectors contain a field called the requestor’s privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates the selector.
- An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment the processor is currently executing. The CPL changes as control is transferred to segments with differing DPLs.

Figure 4-1 shows how these levels of privilege can be interpreted as layers of protection. The center is for the segments containing the most critical software, usually the kernel of the operating system. Outer layers are for the segments of less critical software.

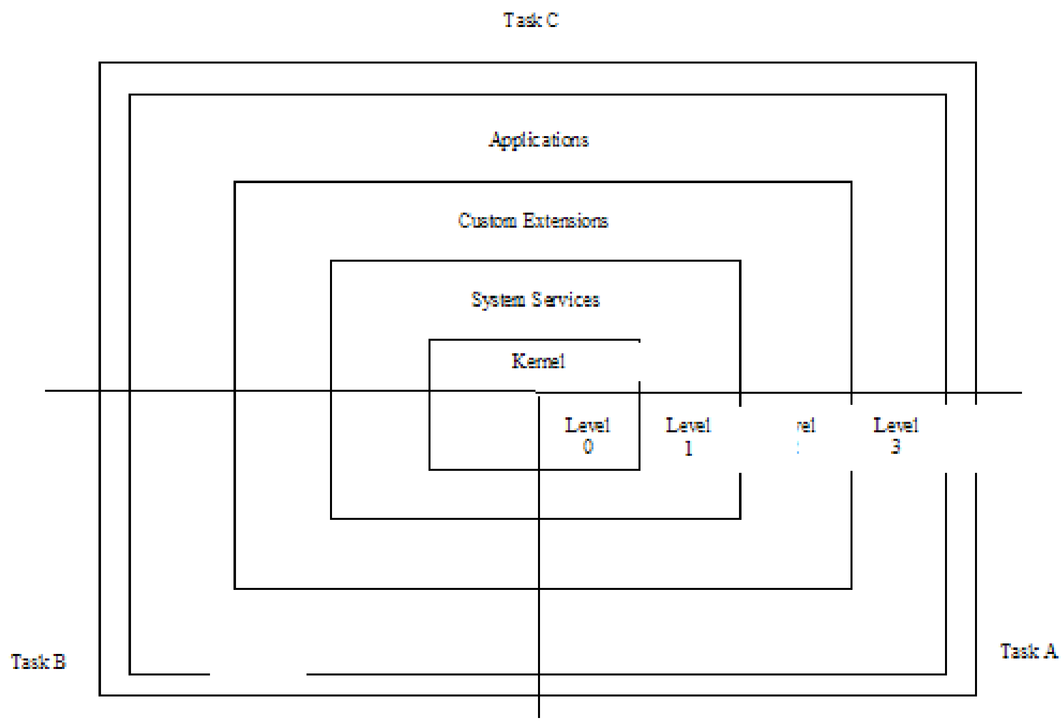


Figure 4-1: Levels of Privilege

System x: The System x servers are powered by Intel processors. Intel processors provide four execution modes, identified with processor privilege levels 0 through 3. The highest privilege level execution mode corresponds to processor privilege level 0; the lowest privilege level execution mode corresponds to processor privilege level 3. The SLES kernel, as with most other UNIX-variant kernels, utilizes only two of these execution modes. The highest, with processor privilege level of 0, corresponds to the kernel mode; the lowest, with processor privilege of 3, corresponds to the user mode.

System p: The System p servers are powered by PowerPC, POWER5 and POWER5+ processors. These processors provide three execution modes, identified by the PR bit (bit 49) and the HV bit (bit 3) of the Machine State Register of the processor. Values of 0 for both PR and HV bits indicate a hypervisor execution mode. An HV bit value of 1, and a PR bit value of 0, indicate a supervisor, or kernel, execution mode. An HV bit value of 1 and a PR bit value of 1 indicate a user execution mode.

System z: The System z systems also provide two execution modes identified by the Problem State bit (bit 15) of the processor's Program Status Word (PSW). A value of 0 indicates a supervisor, or kernel, execution mode, and the value of 1 indicates a problem state, or user, execution mode.

eServer 326: The eServer 326 servers are powered by AMD Opteron processors. These processors provide four execution modes identified with processor privilege levels 0 through 3. The highest privilege level execution mode corresponds to processor privilege level 0; the lowest privilege level execution mode corresponds to processor privilege level 3. The SLES kernel, as with most other UNIX-variant kernels, only utilizes two of these execution modes. The highest, with processor privilege level of 0, corresponds to the kernel mode; the lowest, with processor privilege of 3, corresponds to the user mode.

User and kernel modes, which are offered by all of the eServer systems, implement hardware privilege as follows:

- When the processor is in kernel mode, the program has hardware privilege because it can access and modify any addressable resources, such as memory, page tables, I/O address space, and memory management registers. This is not possible in the user mode.

- When the processor is in kernel mode, the program has hardware privilege because it can execute certain privileged instructions that are not available in user mode.

Thus, any code that runs in kernel mode executes with hardware privileges. Software that runs with hardware privileges includes:

- The base SLES kernel. This constitutes a large portion of software that performs memory management file I/O and process management.
- Separately loaded kernel modules, such as ext3 device driver modules. A module is an object file whose code can be linked to, and unlinked from, the kernel at runtime. The module code is executed in kernel mode on behalf of the current process, like any other statically-linked kernel function.

All other software on the system normally runs in user mode, without hardware privileges, including user processes such as shells, networking client software, and editors. User-mode processes run with hardware privileges when they invoke a system call. The execution of the system call switches the mode from user to kernel mode, and continues operation at a designated address within the kernel where the code of the system call is located.

4.1.2 Software privilege

Software privilege is implemented in the SLES software and is based on the user ID of the process. Processes with user ID of 0 are allowed to bypass the system's access control policies. Examples of programs running with software privilege are:

- Programs that are run by the system, such as the `cron` and `at` daemon.
- Programs that are run by trusted administrators to perform system administration.
- Programs that run with privileged identity by executing setuid programs.

The SLES kernel also has a framework for providing software privilege through *capabilities*. These capabilities, which are based on the POSIX.1e draft, allow breakup of the kernel software privilege associated with user ID zero into a set of discrete privileges based on the operation being attempted. For example, if a process is trying to create a device special file by invoking the `mknod()` system call, instead of checking to ensure that the user ID is zero, the kernel checks to ensure that the process is "capable" of creating device special files. In the absence of special kernel modules that define and use capabilities, as is the case with the TOE, capability checks revert back to granting kernel software privilege based on the user ID of the process.

All software that runs with hardware privileges or software privileges and that implements security enforcing functions is part of the TSF. All other programs are either unprivileged software that run with the identity of the user that invoked the program, or software that executes with privileges but does not implement any security functions. In a properly administered system, unprivileged software is subject to the system's security policies and does not have any means of bypassing the enforcement mechanisms. This unprivileged software need not be trusted in any way and is thus referred to as untrusted software. Trusted processes that do not implement any security function need to be protected from unauthorized tampering using the security functions of the SLES. They need to be trusted to not perform any function that violates the security policy of the SLES.

SLES implements an access control model that enforces Discretionary Access Control and optional additional access control checks implemented in a kernel module known as an Linux Security Module (LSM), such as AppArmor. Discretionary Access Control (DAC) is applied first, and the optional additional checks are applied if and only if the DAC check grants access. AppArmor, if loaded, can only further restrict access, never grant additional access. If access is granted by DAC policy and the AppArmor LSM is loaded, the AppArmor LSM goes through a multi-step process, described in Section 5.8, to determine if access should be allowed.

4.1.2.1 DAC

The DAC model allows the owner of the object to decide who can access that object, and in what manner. Like any other access control model, DAC implementation can be explained by which subjects and objects are under the control of the model, security attributes used by the model, access control and attribute transition rules, and the override (software privilege) mechanism to bypass those rules.

4.1.2.1.1 Subjects and objects

Subjects in SLES are regular processes and kernel threads. They are both represented by the `task_struct` structure. Kernel threads run only in the kernel mode, and are not constrained by the DAC policy. All named objects such as regular files, character and block files, directories, sockets, and IPC objects are under the control of the DAC policy.

4.1.2.1.2 Attributes

Subject attributes used to enforce DAC policy are the process UID, GID, supplementary groups, and process capabilities. These attributes are stored in the `task_structure` of the process, and are affected by the system calls as described in Section 5.2. Object attributes used to enforce DAC policy are owner, group owner, permission bits, and POSIX.1e Access Control Lists (ACLs). These attributes are stored in-core and, for appropriate disk-based file systems, in the on-disk inode.

4.1.2.1.3 Access control rules

DAC access control rules specify how a certain process with appropriate DAC security attributes can access an object with a set of DAC security attributes. In addition, DAC access control rules also specify how subject and object security attributes transition to new values and under what conditions. DAC access control lists are described in detail in Section 5.1.5.

4.1.2.1.4 Software privilege

Software privilege for DAC policy is based on the user ID of the process. At any time, each process has an effective user ID, an effective group ID, and a set of supplementary group IDs. These IDs determine the privileges of the process. A process with a user ID of 0 is a privileged process, with capabilities of bypassing the access control policies of the system. The user name root is commonly associated with user ID 0, but there can be other users with this ID.

Additionally, the SLES kernel has a framework for providing software privilege for DAC policy through capabilities. These capabilities, which are based on the POSIX.1e draft, allow breakup of the kernel software privilege associated with user ID zero into a set of discrete privileges based on the operation being attempted.

For example, if a process is trying to create a device special file by invoking the `mknod()` system call, then instead of checking to ensure that the user ID is zero, the kernel checks to ensure that the process is capable of creating device special files. In the absence of special kernel modules that define and use capabilities, as is the case with the TOE, capability checks revert back to granting kernel software privilege based on the user ID of the process.

4.1.2.2 AppArmor

With AppArmor, it is the system security policy/administrator, unlike the owner in DAC, that controls which files subjects should be allowed to access. AppArmor is implemented as a Linux Security Module (LSM) and is an optionally loaded component of SLES. AppArmor is not required to enforce the security functionality required by the Controlled Access Protection Profile and can only add additional restrictions.

4.1.2.3 Programs with software privilege

Examples of programs running with software privilege are:

- Programs that are run by the system, such as the `cron` and `init` daemons.
- Programs that are run by trusted administrators to perform system administration.
- Programs that run with privileged identity by executing setuid programs.

All software that runs with hardware privileges or software privileges, and that implements security enforcing functions, is part of the TOE Security Functions (TSF). All other programs are either unprivileged programs that run with the identity of the user that invoked the program, or software that executes with privileges but does not implement any security functions.

In a properly administered system, unprivileged software is subject to the security policies of the system and does not have any means of bypassing the enforcement mechanisms. This unprivileged software need not be trusted in any way, and is thus referred to as untrusted software. Trusted processes that do not implement any security function need to be protected from unauthorized tampering using the security functions of the SLES. They need to be trusted to not perform any function that violates the security policy of the SLES.

4.2 TOE Security Functions software structure

This section describes the structure of the SLES software that constitutes the TOE Security Functions (TSF). The SLES system is a multi-user operating system, with the kernel running in a privileged hardware mode, and the user processes running in user mode. The TSF includes both the kernel software and certain trusted non-kernel processes.

Figure 4-2 depicts the TSF and non-TSF portions of software. Subsequent sections provide more detailed descriptions of the kernel and non-kernel TSF architectural subsystems.

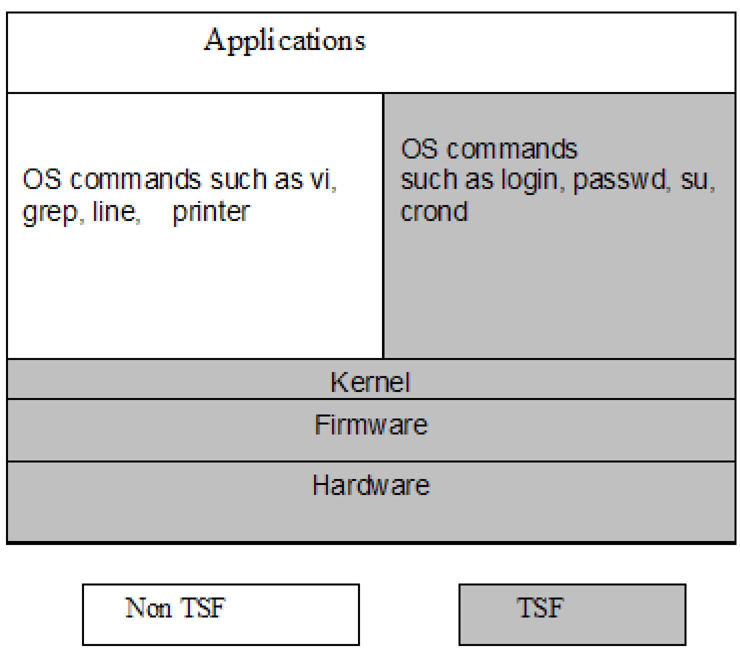


Figure 4-2: TSF and non-TSF software

The concept of breaking the TOE product into logical subsystems is described in the Common Criteria. These logical subsystems are the building blocks of the TOE, and are described in the Functional Descriptions chapter of this paper. They include logical subsystems and trusted processes that implement security functions. A logical subsystem can implement or support one or more functional components. For example, the File and I/O subsystem is partly implemented by functions of the Virtual Memory Manager.

4.2.1 Kernel TSF software

The kernel is the core of the operating system. It interacts directly with the hardware, providing common services to programs, and prevents programs from directly accessing hardware-dependent functions. Services provided by the kernel include the following:

- Control of the execution of processes by allowing their creation, termination or suspension, and communication. These include:
    - Fair scheduling of processes for execution on the CPU.
    - Share of processes in the CPU in a time-shared manner.
    - CPU execution of a process.
    - Kernel suspension when its time quantum elapses.
    - Kernel schedule of another process to execute.
    - Later kernel rescheduling of the suspended process.
- Allocation of the main memory for an executing process. These include:
    - Kernel allowance of processes to share portions of their address space under certain conditions, but protection of the private address space of a process from outside tampering.
    - If the system runs low on free memory, the kernel frees memory by writing a process temporarily to secondary memory, or a swap device.
    - Coordination with the machine hardware to set up a virtual-to-physical address that maps the compiler-generated addresses to their physical addresses.
- File system maintenance. These include:
    - Allocation of secondary memory for efficient storage and retrieval of user data.
    - Allocation of secondary storage for user files.
    - Reclamation of unused storage.
    - Structure of the file system in a well-understood manner.
    - Protection of user files from illegal access.
- Allowance of processes' controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.
- Mediation of access between subjects and objects, allowing controlled access based on DAC and (optionally) AppArmor policy.

The SLES kernel is a fully preemptible kernel. In non-preemptive kernels, kernel code runs until completion. That is, the scheduler is not capable of rescheduling a task while it is in the kernel. Moreover, the kernel code is scheduled cooperatively, not preemptively, and it runs until it finishes and returns to user-space, or explicitly blocks. In preemptive kernels, it is possible to preempt a task at any point, so long as the kernel is in a state in which it is safe to reschedule.

4.2.1.1 Logical components

The kernel consists of logical subsystems that provide different functionalities. Even though the kernel is a single executable program, the various services it provides can be broken into logical components. These components interact to provide specific functions.

Figure 4-3 schematically describes logical kernel subsystems, their interactions with each other, and with the system call interface available from user space.

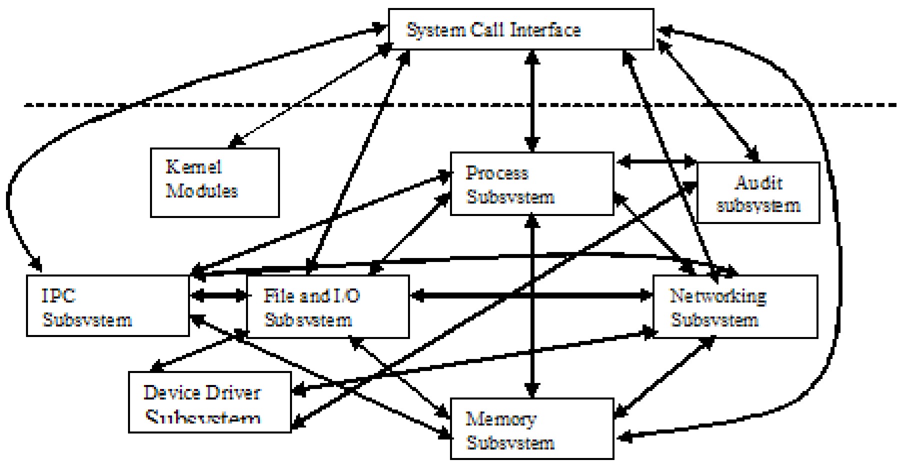


Figure 4-3: Logical kernel subsystems and their interactions

The kernel consists of the following logical subsystems:

- File and I/O subsystem: This subsystem implements functions related to file system objects. Implemented functions include those that allow a process to create, maintain, interact, and delete file-system objects. These objects include regular files, directories, symbolic links, hard links, device-special files, named pipes, and sockets.
- Process subsystem: This subsystem implements functions related to process and thread management. Implemented functions include those that allow the creation, scheduling, execution, and deletion of process and thread subjects.
- Memory subsystem: This subsystem implements functions related to the management of memory resources of a system. Implemented functions include those that create and manage virtual memory, including management of page tables and paging algorithms.
- Networking subsystem: This subsystem implements UNIX and Internet domain sockets, as well as algorithms for scheduling network packets.
- IPC subsystem: This subsystem implements functions related to IPC mechanisms. Implemented functions include those that facilitate controlled sharing of information between processes, allowing them to share data and synchronize their execution in order to interact with a common resource.
- Kernel modules subsystem: This subsystem implements an infrastructure to support loadable modules. Implemented functions include those that load, initialize, and unload kernel modules.
- Device driver subsystem: This subsystem implements support for various hardware and software devices through a common, device-independent interface.

- Audit subsystem: This subsystem implements functions related to recording of security-critical events on the system. Implemented functions include those that trap each system call to record security critical events and those that implement the collection and recording of audit data.

4.2.1.2 Execution components

The execution components of the kernel can be divided into three components: base kernel, kernel threads, and kernel modules depending on their execution perspective.

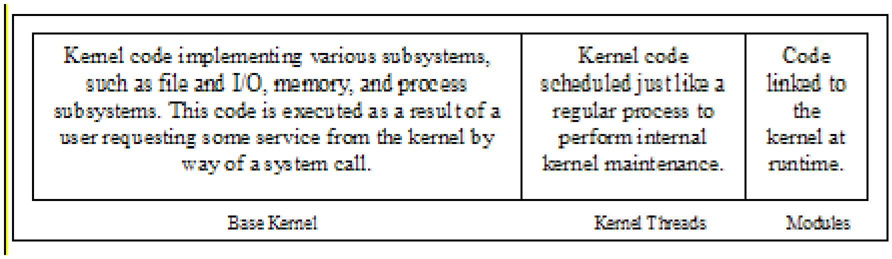


Figure 4-4: Kernel execution components

4.2.1.2.1 Base kernel

The base kernel includes the code that is executed to provide a service, such as servicing a user's system call invocation, or servicing an interrupt or exception event. A majority of the compiled kernel code falls under this category.

4.2.1.2.2 Kernel threads

In order to perform certain routine tasks such as flushing disk caches, reclaiming memory by swapping out unused page frames, the kernel creates internal processes, or threads.

Threads are scheduled just like regular processes, but they do not have context in user mode. Kernel threads execute specific C kernel functions. Kernel threads reside in kernel space, and only run in the kernel mode. Following are some of the kernel threads:

- `keventd` is a process context bottom-half handler that executes tasks created by interrupt handlers, which are queued in the scheduler task queue.
- `kapmd` is a special idle task that handles the events related to Advanced Power Management.
- `kswapd` is a kernel swap daemon responsible for reclaiming pages when memory is running low. The physical page allocator awakens it when the number of free pages for a memory zone falls below a specific threshold.
- `pdflush` is a kernel thread that periodically flushes "dirty" buffers to disk based on a timer. Multiple pdflush threads may run up to the maximum tunable by `/proc/sys/vm/nr_pdflush_threads`.
- `kjournald` is a process that manages the logging device journal, periodically commits the current state of the file system to disk, and reclaims space in the log by flushing buffers to disk.
- Kernel threads are created with a call to `kernel_thread()`, and users can list them with the `ps axu` command. The command shows the kernel threads in square brackets, and can be recognized by their virtual memory size (`VSZ`) of 0; an example is `[kjournald]`.

4.2.1.2.3 Kernel modules and device drivers

Kernel modules are pieces of code that can be loaded and unloaded into and out of the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. Once loaded, the kernel module object code can access other kernel code and data in the same manner as statically-linked kernel object code.

A device driver is a special type of kernel module that allows the kernel to access the hardware connected to the system. These devices can be a hard disk, monitor, or network interface. The driver interacts with the remaining part of the kernel through a specific interface, which allows the kernel to deal with all devices in a uniform way, independently of their underlying implementations.

4.2.2 Non-kernel TSF software

The non-kernel TSF software consists of trusted programs that are used to implement security functions. Note that shared libraries, including PAM modules in some cases, are used by trusted programs. The trusted commands can be grouped as follows.

- Daemon processes that do not directly run on behalf of a user, but are started at system startup or upon demand of a system administrator. Daemon processes are responsible for setting the appropriate user identity when performing a service on behalf of a user. Following are the daemon processes that provide TSF functionality.
    - The **atd** daemon is the server that reads at jobs submitted by all users and performs tasks specified in them on behalf of the user. atd is started by the `init` program during system initialization.
    - The **auditd** daemon reads audit records from the kernel buffer through the audit device and writes them to disk in the form of audit logs.
    - The **cron** daemon is the daemon that reads the `crontab` files for all users and performs tasks specified in the `crontab` files on behalf of the user. The init program starts the `cron` daemon during system initialization. The `crontab` file and `cron` daemon are the client-server pair that allow the execution of commands on a recurring basis at a specified time.
    - The **init** program is the userspace process that is ancestor to all other userspace processes. It starts processes as specified in the `/etc/inittab` file.
    - The **sshd** daemon is the program for secure shell. The `ssh` command and `sshd` daemon are the client-server pair that allow authorized users to log in from remote systems using secure encrypted communications.
    - The **vsftpd** daemon is the Very Secure File Transfer Protocol daemon that allows authorized users to transfer files to and from remote systems.
    - The **xinetd** daemon accepts incoming network connections and dispatches the appropriate child daemon to service each connection request.
- Following are programs that are executed by an unprivileged user and need access to certain protected databases to complete their work.
    - The **at** program is the program used by all users to submit tasks to be performed at a later time.
    - The **atrm** program removes jobs already queued for execution. `atrm` deletes jobs, whose job numbers are passed to the command line as arguments.
    - The **chage** command allows the system administrator to change the user password expiry information. Refer to the `chage` man page for more detailed information.

  - The **crontab** program is the program used to install, deinstall, or list the tables used to drive the `cron` daemon. Users can have their own `crontab` files that set up the time and frequency of execution, as well as the command or script to execute.
  - The **gpasswd** command administers the `/etc/group` file and `/etc/gshadow` file if compiled with SHADOWGRP defined. The `gpasswd` command allows system administrators to designate group administrators for a particular group. Refer to the `gpasswd` man page for more detailed information.
  - The **login** program is used when signing on to a system. If root is trying to log in, the program makes sure that the login attempt is being made from a secure terminal listed in `/etc/securetty`. The login program prompts for the password and turns off the terminal echo in order to prevent the password from being displayed as the user types it. The login program then verifies the password for the account; although three attempts are allowed before login dies, the response becomes slower after each failed attempt. Once the password is successfully verified, various password aging restrictions, which are set in the `/etc/login.defs` file, are checked. If the password age is satisfactory, then the program sets the user ID and group ID of the process, changes the current directory to the user's home directory, and executes a shell specified in the `/etc/passwd` file. Refer to the `login` man page for more detailed information.
  - The **passwd** command updates a user's authentication tokens, and is configured to work through the PAM API. It then configures itself as a password service with PAM, and uses configured password modules to authenticate and then update a user's password. The `passwd` command turns off terminal echo while the user is typing the old as well as the new password, in order to prevent displaying the password typed by the user. Refer to the `passwd` man page for more detailed information.
  - The **su** command allows a user to run a shell with substitute user and group IDs. It changes the effective user and group IDs to those of the new user. Refer to the `su` man page for more detailed information.
- The following are trusted programs that do not fit into the above 2 categories.
  - The alternative Linux form of `getty`, **agetty** opens a tty port, prompts for a login name, and invokes the `/bin/login` command. The `/sbin/init` program invokes it when the system becomes available in a multi-user mode.
  - The **amtu** program is a special tool provided to test features of the underlying hardware that the TSF depends on. The test tool runs on all hardware architectures that are targets of evaluation and reports problems with any underlying functionalities.
  - In addition to setting the audit filter rules and watches on file system objects, **auditctl** can be used to control the audit subsystem behavior in the kernel when `auditd` is running. Only an administrative user is allowed to use this command.
  - The **ausearch** command finds audit records based on different criteria from the audit log. Only an administrative user is allowed to use this command.
  - **aureport** produces reports of the audit system logs.
  - The **init** program is the first program to run after the kernel starts running. It is the parent of all processes, and its primary role is to create processes from a script stored in the /etc/inittab file. This file usually has entries that cause `init` to spawn `getty` on each line that users can log in.
  - The **chsh** command allows users to change their login shells. If a shell is not given on the command line, `chsh` prompts for one.

- The **chfn** command allows users to change their finger information. The finger command displays that information, which is stored in the `/etc/passwd` file.
- The **date** command is used to print or set the system date and time. Only an administrative user is allowed to set the system date and time.
- The **groupadd**, **groupmod**, and **groupdel** commands allow an administrator to add, modify, or delete a group, respectively. Refer to their respective man pages for more detailed information.
- The **hwclock** command is used to query and set the hardware clock. Only an administrative user is allowed to set the system hardware clock.
- The minimal form of `getty`, **mingetty** is for consoles, and provides the same functionality as `agetty`. However, unlike `agetty`, which is used for serial lines, `mingetty` is used for virtual consoles.
- The **newgrp** command logs into another groupid.
- The **openssl** program is a command-line tool for using the various cryptography functions of the Secure Socket Layer (SSL v3) and Transport Layer Security (TSL v1) network protocols.
- **pam_tally** manages the `/var/log/faillog` file to reset the failed login counter.
- The **ping** and **ping6** commands, for IPv4 and IPv6 respectively, use the mandatory `ECHO_REQUEST` datagram of the Internet Control Message Protocol (ICMP) to elicit an `ICMP_ECHO_RESPONSE` from a host or a gateway.
- The **ssh** command is a program for logging into a remote machine and for executing commands on a remote machine. It provides secure encrypted communications between two untrusted hosts over an insecure network.
- **star** is a version of the tar command that preserves extended attributes. Extended attributes are the means by which ACLs are associated with file system objects.
- The **stunnel** program is designed to work as an SSL encryption wrapper between remote clients and local or remote servers.
- The **useradd**, **usermod**, and **userdel** commands allow an administrator to add, modify, or delete a user account, respectively. Refer to their respective man pages for more detailed information.
- **unix_chkpwd** is the helper program for the `pam_unix` PAM module that checks the validity of passwords at login time. It is not designed to be directly executed.

4.3 TSF databases

Section 6.2.8.5 of the Security Target identifies the primary TSF databases used in SLES and their purposes. These are listed either as individual files, by pathname, or as collections of files.

With the exception of databases listed with the User attribute (which indicates that a user can read, but not write, the file), all of these databases are only accessible to administrative users. None of these databases is modifiable by a user other than an administrative user. Access control is performed by the file system component of the SLES kernel. For more information about the format of these TSF databases, please refer to their respective section of man pages.

See section 6.2.8.5 in the Security Target.

4.4 Definition of subsystems for the CC evaluation

Previous sections of this paper defined various logical subsystems that constitute the SLES system. One of these logical subsystems alone can provide, or two or more can combine to provide, security functionalities.

This section briefly describes the functional subsystems that implement the required security functionalities and the logical subsystems that are part of each of the functional subsystems.

The subsystems are structured into those implemented within the SLES kernel, and those implemented as trusted processes.

4.4.1 Hardware

The hardware consists of the physical resources such as CPU, main memory, registers, caches, and devices that effectively make up the computer system. Chapter 3 details the various hardware architectures supported in this evaluation.

4.4.2 Firmware

The firmware consists of the software residing in the hardware that is started when the system goes through a power-on reset. In addition to initializing the hardware and starting the operating system, on the partitioning-capable platforms the firmware provides LPAR support as well.

4.4.3 Kernel subsystems

This section describes the subsystems implemented as part of the SLES kernel.

- File and I/O: This subsystem includes only the file and I/O management kernel subsystem.
- Process control: This subsystem includes the process control and management kernel subsystem.
- Inter-process communication: This subsystem includes the IPC kernel subsystem.
- Networking: This subsystem contains the kernel networking subsystem.
- Memory management: This subsystem contains the kernel memory management subsystem.
- Kernel modules: This subsystem contains routines in the kernel that create an infrastructure to support loadable modules.
- Device drivers: This subsystem contains the kernel device driver subsystem.
- Audit: This subsystem contains the kernel auditing subsystem.

4.4.4 Trusted process subsystems

This section describes the subsystems implemented as trusted processes.

- System initialization: This subsystem consists of the boot loader (GRUB, LILO, Yaboot, or z/IPL) and the `init` program.
- Identification and authentication: This subsystem contains the `su`, `passwd`, and `login` trusted commands, as well as the `agetty` trusted process. This subsystem also includes PAM shared library modules.
- Network applications: This subsystem contains `vsftpd` and `sshd` trusted processes, which interact with PAM modules to perform authentication. It also includes the `ping` program.
- Batch processing: This subsystem contains the trusted programs used for the processing of batch jobs. They are `crontab` and `cron and at` and `atd`.
- System management: This subsystem contains the trusted programs used for system management activities. Those include the following programs:

- `gpasswd`
- `chage`
- `useradd, usermod, userdel`
- `groupadd, groupmode, groupdel`
- `chsh`
- `chfn`
- `openssl`

4.4.5 User-level audit subsystem

This subsystem contains the portion of the audit system that lies outside the kernel. This subsystem contains the `auditd` trusted process, which reads audit records from the kernel buffer, and transfers them to on-disk audit logs, the `ausearch` trusted search utility, the `autrace` trace utility, the audit configuration file, and audit libraries.

5 Functional descriptions

The kernel structure, its trusted software, and its Target of Evaluation (TOE) Security Functions (TSF) databases provide the foundation for the descriptions in this chapter.

5.1 File and I/O management

The file and I/O subsystem is a management system for defining objects on secondary storage devices. The file and I/O subsystem interacts with the memory subsystem, the network subsystem, the inter-process communication (IPC) subsystem, the process subsystem, and the device drivers.

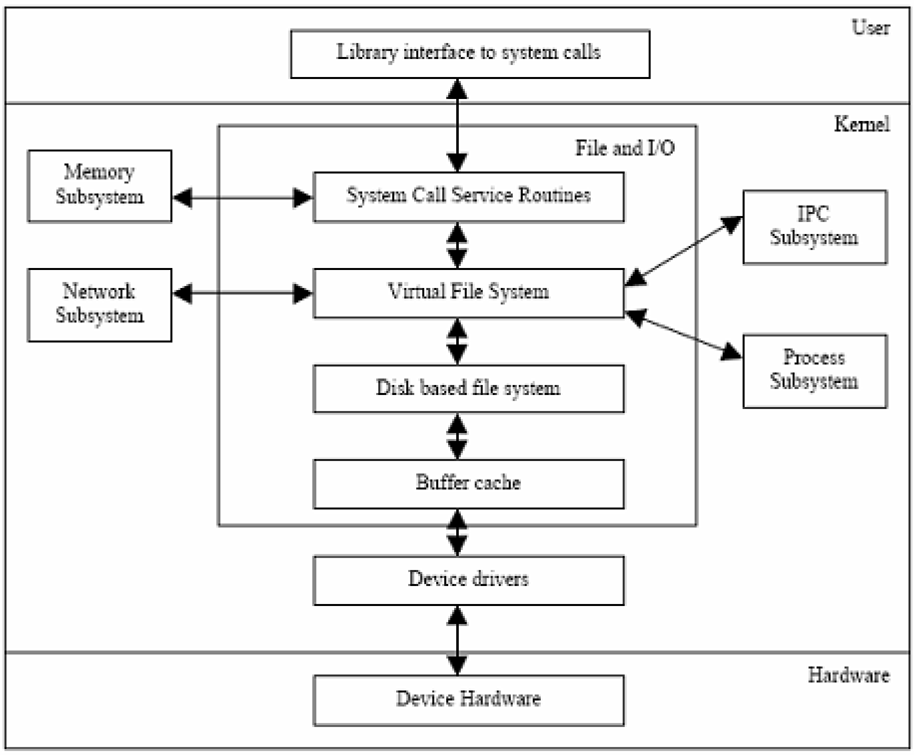


Figure 5-1: File and I/O subsystem and its interaction with other subsystems

A file system is a container for objects on the secondary storage devices. The implementation of the file system allows for the management of a variety of types of file systems. The file systems supported by TOE are ext3, proc, tmpfs, sysfs, devpts, CD-ROM, rootfs, and binfmt_misc.

At the user-interface level, a file system is organized as a tree with a single root, called a directory. A directory contains other directories and files, which are the leaf nodes of the tree. Files are the primary containers of user data. Additionally, files can be symbolic links, named pipes, sockets, or special files that represent devices.

This section briefly describes the SLES file system implementation, and focuses on how file system object attributes support the kernel's implementation of the Discretionary Access Checks (DAC) policy of the kernel. This section also highlights how file system data and metadata are allocated and initialized to satisfy the object reuse requirement.

In order to shield user programs from the underlying details of different types of disk devices and disk-based file systems, the SLES kernel provides a software layer that handles all system calls related to a standard UNIX file system. This common interface layer, called the Virtual File System, interacts with disk-based file systems whose physical I/O devices are managed through device special files.

This section of this paper is divided into three subsections: Virtual File System, Disk-Based File Systems, and Discretionary Access Control. The subsections describe data structures and algorithms that comprise each subsystem, with special focus on access control and allocation mechanisms.

5.1.1 Virtual File System

The Virtual File System (VFS) provides a common interface to users for performing all file-related operations, such as open, read, write, change owner, and change mode. The key idea behind the VFS is the concept of the common file model, which is capable of representing all supported file systems.

For example, consider a SLES system where an ext3 file system is mounted on the ext3mnt directory, and a CD-ROM file system is mounted on the cdmnt directory, as in Figure 5-2.

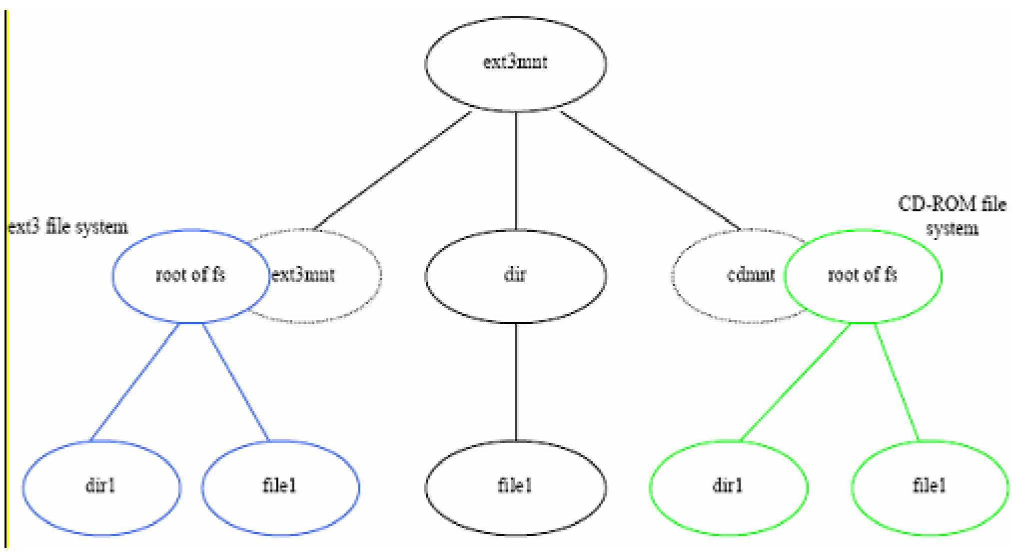


Figure 5-2: ext3 and CD-ROM file systems before mounting

To a user program, the virtual file system appears as follows:

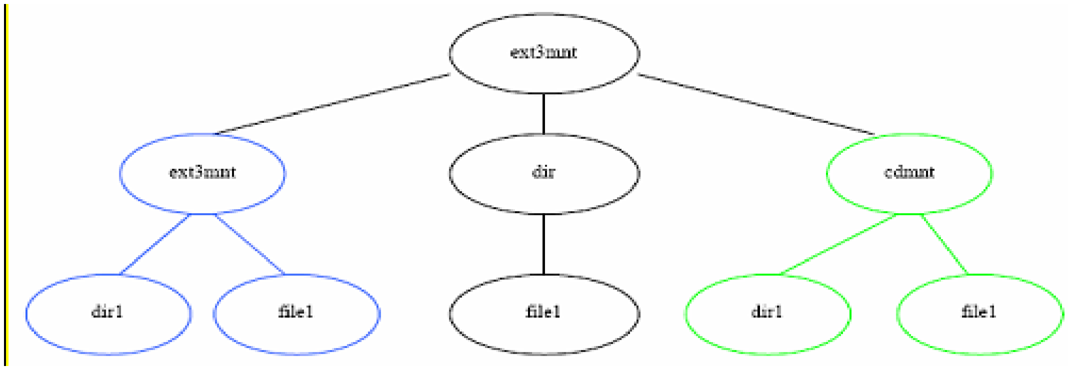


Figure 5-3: ext3 and CD-ROM file systems after mounting

The root directory is contained in the root file system, which is ext3 in this TOE. All other file systems can be mounted on subdirectories of the root file system.

The VFS allows programs to perform operations on files without having to know the implementation of the underlying disk-based file system. The VFS layer redirects file operation requests to the appropriate file system-specific file operation. An example is in Figure 5-4.

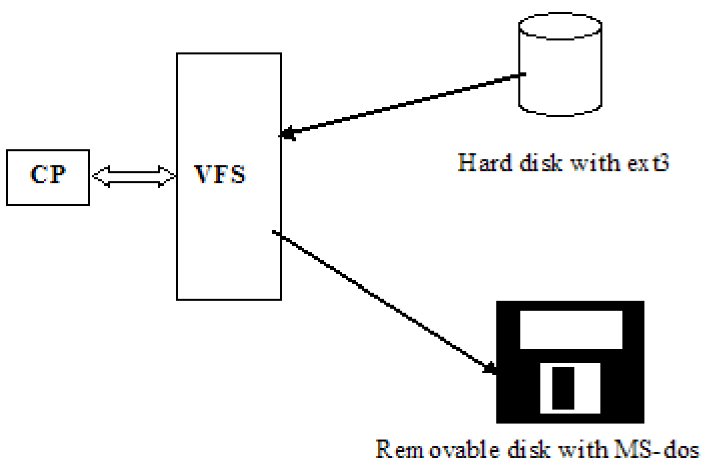


Figure 5-4: Virtual file system

Almost all of the system call interfaces available to a user program in the common file model of VFS involve the use of a file pathname. The file pathname is either an absolute pathname such as `/ext3mnt/file1`, or a relative pathname such as `ext3mnt/file1`. The translation of a pathname to file data is relevant to security, because the kernel performs access checks as part of this translation mechanism.

The following list describes the security-relevant data structures of the VFS.

`super_block`: Stores information about each mounted file system, such as file system type, block size, maximum size of files, and `dentry` object (described below) of the mount point. The actual data structure in SLES is called `struct super_block`.

inode: Stores general information about a specific file, such as file type and access rights, file owner, group owner, length in bytes, operations vector, time of last file access, time of last file write, and time of last inode change. An inode is associated to each file and is described in the kernel by a struct inode data structure.

file: Stores the interaction between an open file and a process, such as the pointer to a file operation table, current offset (position within the file), user id, group id, and the dentry object associated with the file. A file exists only in kernel memory during the period when each process accesses a file. An open file is described in the SLES kernel by a struct file.

dentry: Stores information about the linking of a directory entry with the corresponding file, such as a pointer to the inode associated with the file, filename, pointer to dentry object of the parent directory, or pointer to directory operations.

vfsmount: Stores information about a mounted file system, such as dentry objects of the mount point and the root of the file system, the name of device containing the file system, and mount flags.

The kernel uses the above data structures while performing pathname translation and file system mounting operations relevant to security.

5.1.1.1 Pathname translation

When performing a file operation, the kernel translates a pathname to a corresponding inode. The pathname translation process performs access checks appropriate to the intended file operation. For example, any file system function that results in a modification to a directory, such as creating a file or deleting a file, checks to make sure that the process has write access to the directory being modified. Directories cannot be directly written into.

Access checking in VFS is performed while an inode is derived from the corresponding pathname. Each access check involves checking DAC policy first, and if access is permitted by DAC policy, then checking the AppArmor policy. Pathname lookup routines break up the pathname into a sequence of file names, and depending on whether the pathname is absolute or relative, the lookup routines start the search from the root of the file system or from the current directory of the process, respectively. The dentry object for this starting position is available through the fs field of the current process.

Using the inode of the initial directory, the code looks at the entry that matches the first name to derive the corresponding inode. Then the directory file that has that inode is read from the disk, and the entry matching the second name is looked up to derive the corresponding inode. This procedure is repeated for each name included in the path. At each file lookup within a directory stage, an access check is made to ensure that the process has appropriate permission to perform the search. The last access check performed depends on the system call.

For example, when a new file is created, an access check is performed to ensure that the process has write access to the directory. If an existing file is being opened for read, a permission check is made to ensure that the process has read access to that file.

The example in Figure 5-5 is a simplified description of a pathname lookup. In reality, the algorithm for lookup becomes more complicated because of the presence of symbolic links, dot (.), dot dot (..) and extra slash (/) characters in the pathname. Even though these objects complicate the logic of the lookup routine, the access check mechanism remains the same.

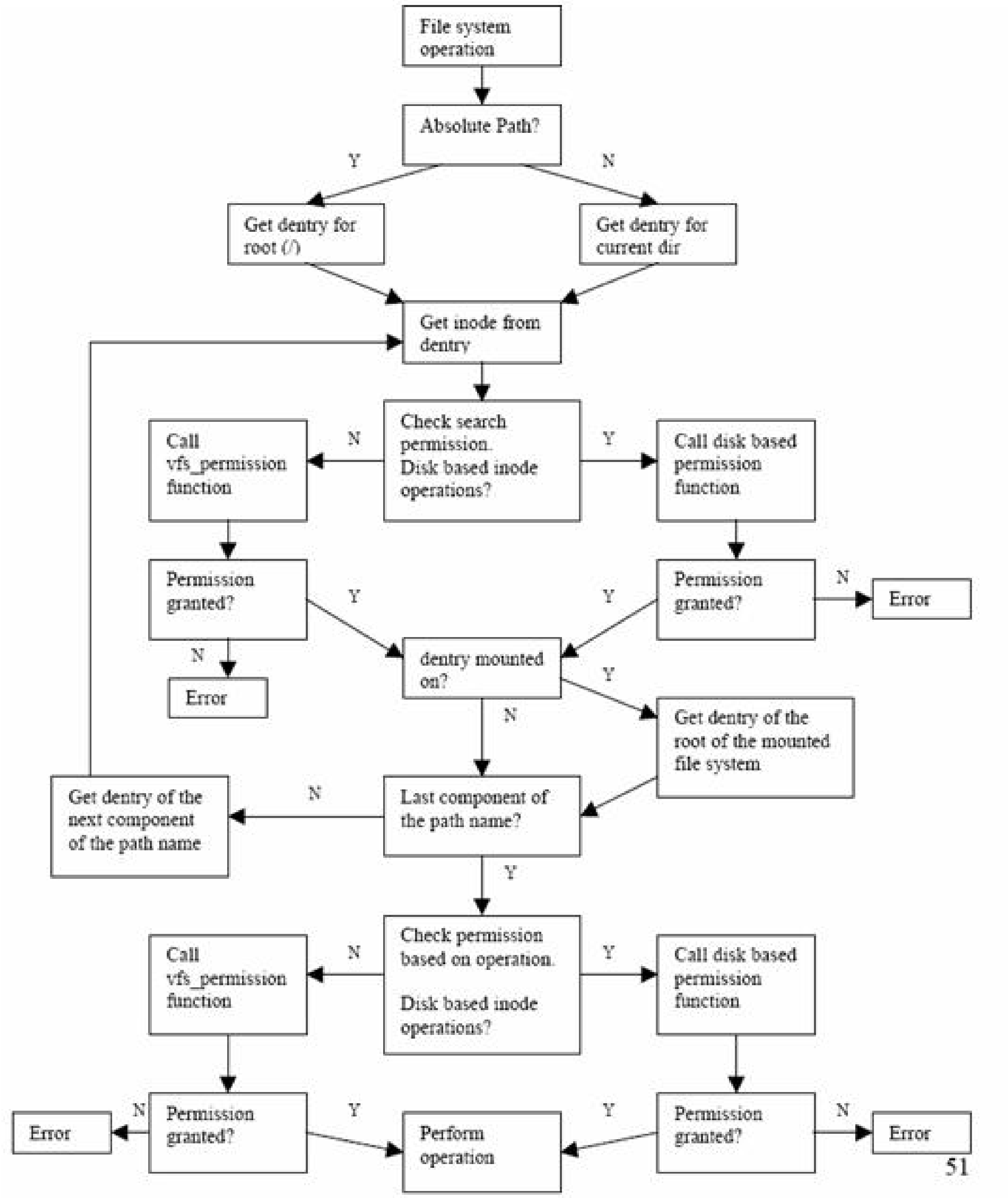


Figure 5-5 VFS pathname translation and access control checks

5.1.1.2 open()

The following describes the call sequence of an `open()` call to create a file:

1. Call the `open()` system call with a relative pathname and flags to create a file for read and write.
2. `open()` calls `open_namei()`, which ultimately derives the `dentry` for the directory in which the file is being created. If the pathname contains multiple directories, search permission for all directories in the path is required to get access to the file.

   This search permission check is performed for each directory `dentry` by calling `permission()`. If the operation vector of the `inode`, which contains pointers to valid `inode` operation routines, is set, then each call to `permission()` is diverted to the disk-based file system-specific permission call. Otherwise, `generic_permission()` is called, to ensure that the process has the appropriate permission. If at this stage the process has the DAC permission, because either the generic or disk-based file system granted the permission, then AppArmor permission is checked through the `security_inode_permission()` LSM call.
3. Once the directory `dentry` is found, `permission()` is called to make sure the process is authorized to write in this directory. Again, if the operation vector of the `inode` is set, then the call to `permission()` is diverted to the disk-based file system-specific permission call; otherwise `generic_permission()` is called to ensure that the process has the appropriate permission. If at this stage the process has the DAC permission, because either the generic or disk-based file system granted the permission, then AppArmor permission is checked through the `security_inode_permission()` LSM call.
4. If the user is authorized to create a file in this directory, then `get_empty_filp()` is called to get a file pointer. `get_empty_filp()` calls `memset()` to ensure that the newly allocated file pointer is zeroed out, thus taking care of the object reuse requirement. To create the file, `get_empty_filp()` calls the disk-based file system-specific open routine through the file operations vector in the file pointer.

At this point, data structures for the file object, `dentry` object, and `inode` object for the newly created file are set up correctly, whereby the process can access the `inode` by following a pointer chain leading from the `file` object to the `dentry` object to the `inode` object. The following diagram shows the simplified linkage:

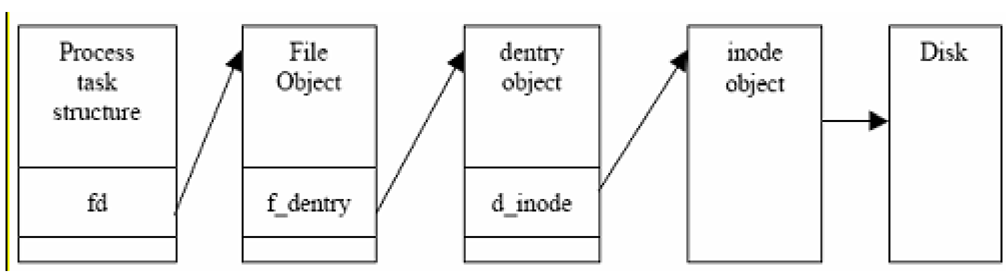


Figure 5-6: VFS data structures and their relationships with each other

5.1.1.3 write()

Another example of a file system operation is a `write()` system call to write to a file that was opened for writing. The `write()` system call in VFS is very straightforward, because access checks have already been performed by `open()`. The following list shows the call sequence of a `write()` call:

1. Call the `write()` system call with the file descriptor that was returned by `open()`.
2. Call `fget()` to get the file pointer corresponding to the file descriptor.
3. If the file operation vector of the file pointer is set, use the `inode` operation vector to call the disk-based file system's `write()` routine of the disk-based file system.

5.1.1.4 mount()

An administrator mounts file systems using the `mount()` system call. The `mount()` system call provides the kernel with the following:

- the file system type
- the pathname of the mount point
- the pathname of the block device that contains the file system
- the flags that control the behavior of the mounted file system
- a pointer to a file system dependent data structure (that may be NULL).

For each mount operation, the kernel saves the mount point and the mount flags in mounted file system descriptors. Each mounted file system descriptor is a `vfsmount` type of data structure. The `sys_mount()` function in the kernel copies the value of the parameters into temporary kernel buffers, acquires the big kernel lock, and invokes the `do_mount()` function to perform the mount.

There are no object reuse issues to handle during file system mounting because the data structures created are not directly accessible to user processes. However, there are security-relevant mount flags that affect access control. Following are the security-relevant mount flags and their implications for access control.

- `MS_RDONLY`: The file system is mounted in read-only mode. Write operations are prohibited for all files regardless of their mode bits.
- `MS_NOSUID`: the kernel ignores suid and sgid bits on executables when executing files from this file system.
- `MS_NODEV`: Device access to a character or block device is not permitted from files on this file system.
- `MS_NOEXEC`: Execution of any programs from this file system is not permitted, even if the execute bit is set for the program binary.
- `MS_POSIXACL`: Indicates if ACLs on files on this file system are to be honored or ignored.

5.1.1.5 Shared subtrees

Shared subtrees have been implemented in VFS. This allows an administrator to configure the way the file system mounts will coexist in the tree, the relationships between them, and how they propagate in different namespaces. This increases flexibility in the way namespaces can be populated and presented to users. For detailed information about the shared-subtree feature, see http://lwn.net/Articles/159077 and http://lwn.net/Articles/159092.

The shared-subtree feature introduces new types of mounts:

- Unbindable Mount: This mount does not forward or receive propagation. This mount type can not be bind-mounted, and it is not valid to move it under a shared mount.
- Slave Mount: A slave mount remains tied to its parent mount and receives new mount or unmount events from there. The mount or unmount events in a slave mount do not propagate elsewhere.
- Shared Mount: When this mount is used, all events generated are automatically propagated to the shared mount subtree. Shared mounts are able to propagate events to others belonging to the same peer group.
- Private Mount: This works as the previous existent mount. Private mounts cannot be propagated to any other mounts, except when forced by administrators using the bind operation. Any kind of mounts can be converted to private mounts.

5.1.2 Disk-based file systems

Disk-based file systems deal with how the data is stored on the disk. Different disk-based file systems employ different layouts and support different operations on them. For example, the CD-ROM file system does not support the write operation. The TOE supports two disk-based file systems: ext3, and the ISO 9660 File System for CD-ROM.

This section looks at data structures and algorithms used to implement these two disk-based file systems and continues the description of open() and write() system calls in the context of disk-based file systems.

5.1.2.1 Ext3 file system

The SLES kernel's ext3 file system kernel is a robust and efficient file system that supports the following:

- Automatic consistency checks
- Immutable files
- Preallocation of disk blocks to regular files
- Fast symbolic links
- ACLs
- Journaling

The file system partitions disk blocks into groups. Each group includes data blocks and inode blocks in adjacent tracks, which allow files to be accessed with a lower average disk seek time. In addition to the traditional UNIX file object attributes such as owner, group, permission bits, and access times, the SLES ext3 file system supports Access Control Lists (ACLs) and Extended Attributes (EAs). ACLs provide a flexible method for granting or denying access, which is granular down to an individual user, directory, or file.

5.1.2.1.1 Extended Attributes

An extended attribute (EA, aka xattr) provides a mechanism for setting special flags on a directory or a file. Some of these improve the usability of the system, while others improve the security of the system. EAs also provide a mechanism that allows persistent storage of security attributes—DAC ACLs.

The EA namespace is partitioned. ACLs make use of reserved namespaces with access restricted to administrative users (and object owner in some cases). Special checks are performed in the xattr syscalls to ensure that only administrative users and privileged system services can access the reserved namespaces. The system.posix_acl_access and system.posix_acl_default namespaces are reserved for ACL metadata. This namespace is restricted to the object owner and is accessible by administrative users.

5.1.2.1.1.1 Access Control Lists

ACLs provide a way of extending directory and file access restrictions beyond the traditional owner, group, and world permission settings. For more details about the ACL format, refer to Discretionary Access Control, Section 5.1.5, of this document, and section 6.2.4.3 of the SLES Security Target document. EAs are stored on disk blocks allocated outside of an inode. Security-relevant EAs provide the following functionality:

- Immutable: if this attribute is set, the file cannot be modified, no link can be created to it, and it cannot be renamed or removed. Only an administrator can change this attribute.
- Append only: if this attribute is set, the file may only be modified in append mode. The append only attribute is useful for system logs.

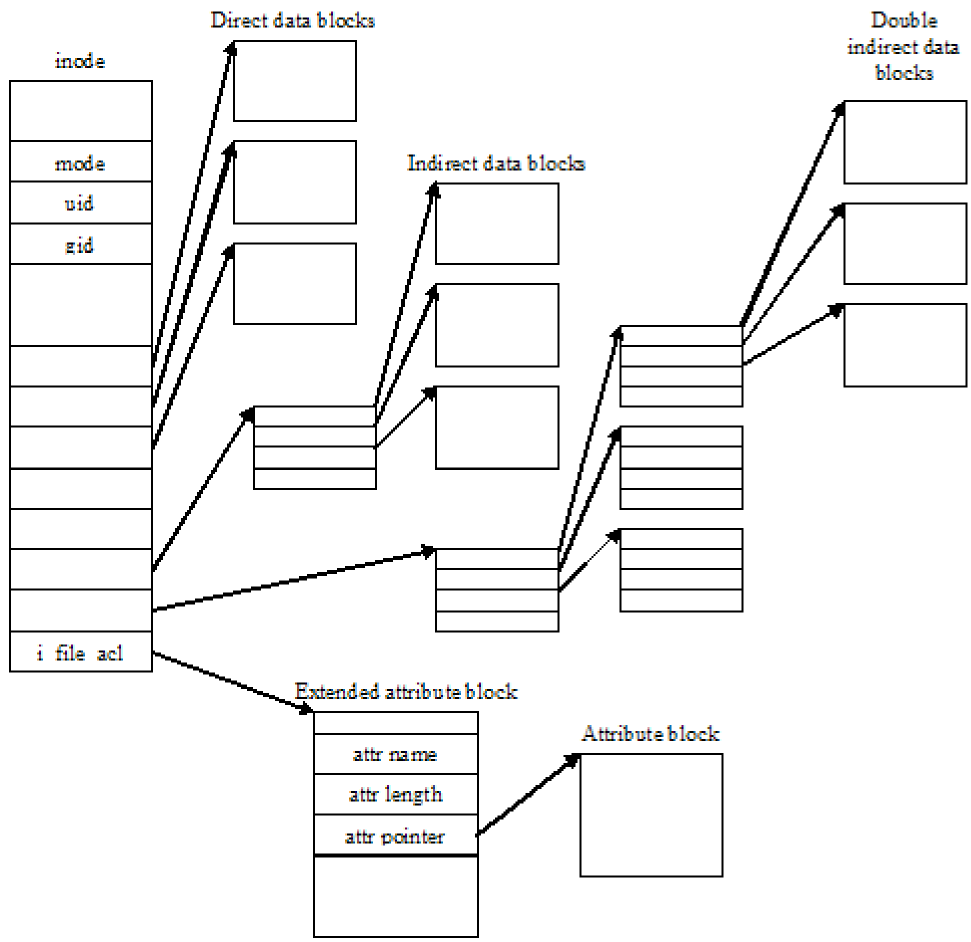


Figure 5-7: Security attributes, extended security attributes, and data blocks for the ext3 inode

5.1.2.1.2 Data structures

The following data structures and `inode` operations illustrate how the ext3 file system performs DAC and object reuse.

- **ext3_super_block**: The on-disk counterpart of the `superblock` structure of VFS, `ext3_super_block` stores file system-specific information such as the total number of `inodes`, block size, and fragment size.

- **ext3_group_desc**: Disk blocks are partitioned into groups. Each group has its own group descriptor. `ext3_group_desc` stores information such as the block number of the `inode` bitmap, and the block number of the block bitmap.
- **ext3_inode**: The on-disk counterpart of the `inode` structure of VFS, `ext3_inode` stores information such as file owner, file type and access rights, file length in bytes, time of last file access, number of data blocks, pointer to data blocks, and file access control list.
- **ext3_xattr_entry**: This structure describes an extended attribute entry. The `ext3_xattr_entry` stores information such as attribute name, attribute size, and the disk block that stores the attribute. ACLs are stored on disk using this data structure, and associated to an `inode` by pointing the `inode`'s `i_file_acl` field to this allocated extended attribute block.
- **ext3_create()**: This routine is called when a file create operation makes a transition from VFS to a disk-based file system. `ext3_create()` starts journaling, and then calls `ext3_new_inode()` to create the new `inode`.
- **ext3_lookup()**: This routine is called when VFS `real_lookup()` calls the disk-based file system lookup routine of the disk-based file system through the `inode` operation vector. The `ext3_find_entry()` is called by `ext3_lookup()` to locate an entry in a specified directory with the given name.
- **ext3_permission()**: This is the entry point for all Discretionary Access Checks (DACs). This routine is invoked when VFS calls to the `permission()` routine are diverted based on the ext3 `inode`'s `inode` operation vector `i_op` of the ext3 `inode`. `ext3_permission()` calls `generic_permission()`.
- **ext3_get_block()**: This is the general-purpose routine for locating data that corresponds to a regular file. `ext3_get_block()` is invoked when the kernel is looking for, or allocating, a new data block. The routine is called from routines set up in the address-space operations vector, `a_ops`, which is accessed through the inode's `i_mapping` field of the inode. `ext3_get_block()` calls `ext3_get_block_handle()`, which in turn calls `ext3_alloc_branch()` if a new data block needs to be allocated. `ext3_alloc_branch()` explicitly calls `memset()` to zero-out the newly allocated block, thus taking care of the object reuse requirement.

Figure 5-8 illustrates how new data blocks are allocated and initialized for an ext3 file.

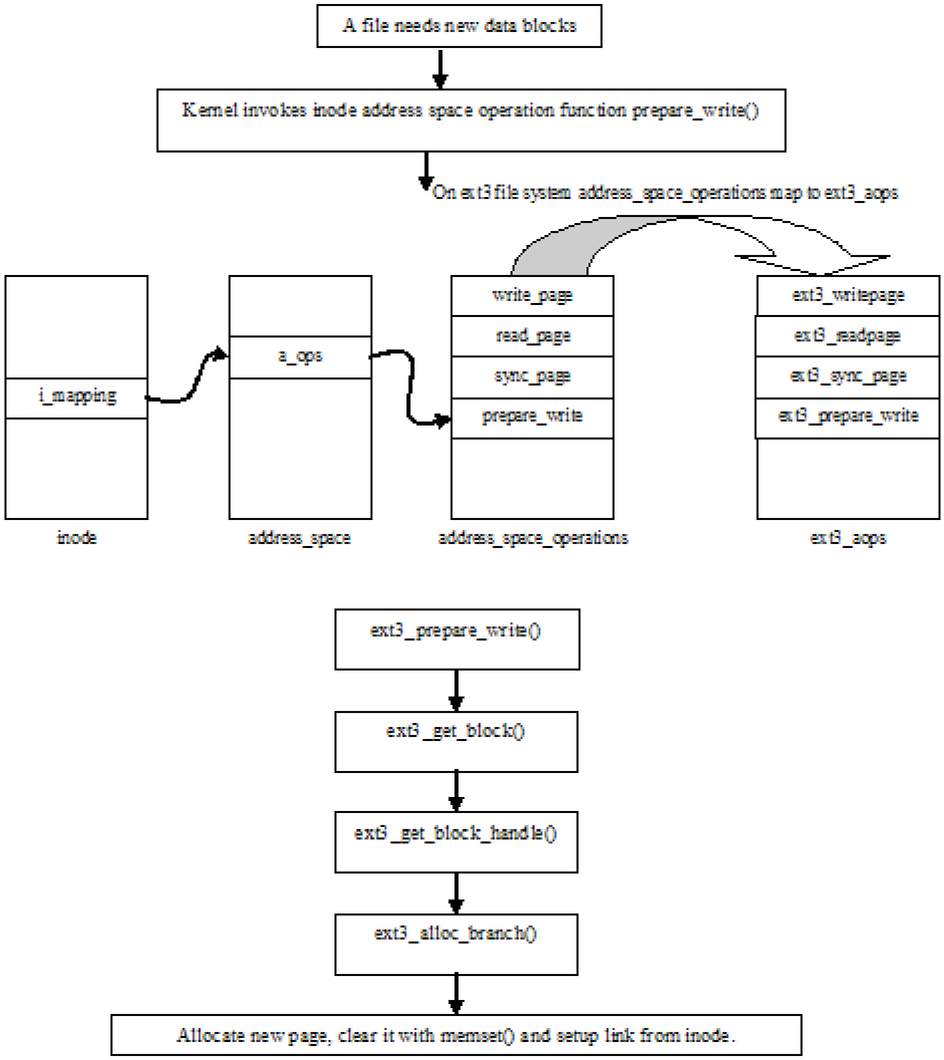


Figure 5-8: New data blocks are allocated and initialized for an ext3 field

Figure 5-9 shows how for a file on the ext3 file system, `inode_operations` map to `ext3_file_inode_operations`.

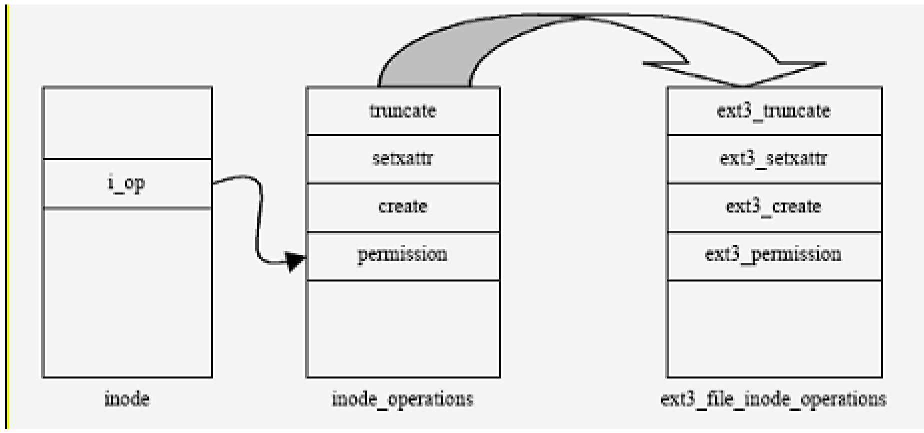


Figure 5-9: Access control on ext3 file system

Similarly, for directory, symlink, and special-file types of objects, `inode` operations map to `ext3_dir_inode_operations`, `ext3_symlink_inode_operations`, and `ext3_special_inode_operations`, respectively.

`ext3_truncate()` is the entry point for truncating a file. The `ext3_truncate()` routine is invoked when VFS calls to the `sys_truncate()` routine are diverted based on the ext3 inode's inode operation vector `i_op` of the ext3 `inode`. This routine prevents the truncation of inodes whose extended attributes mark them as being append-only or immutable.

5.1.2.2 ISO 9660 file system for CD-ROM

The SLES kernel supports the ISO 9660 file system for CD-ROM. Refer to the HOWTO document by Martin Hinner on the Linux Documentation Project Web site for a detailed specification of the ISO 9660 file system: http://www.tldp.org/HOWTO/Filesystems-HOWTO.html.

5.1.2.2.1 Data structures and algorithms

The following data structures and inode operations implement the file system on the SLES kernel.

- vfs_permission(): Because the file system is a read-only file system, there are no object reuse implications with respect to allocating data blocks. The discretionary access check is performed at the VFS layer with the `vfs_permission()` routine, which grants permission based on a process's fsuid field.
- isofs_sb_info: The CD-ROM file system super block isofs_sb_info stores file system-specific information, such as the number of inodes, number of zones, maximum size, and fields for the mount command line option to prohibit the execution of suid programs.
- iso_inode_info: The iso_inode_info is in-core inode information for CD-ROM file objects. iso_inode_info stores information, such as file format, extent location, and a link to the next inode.
- isofs_lookup(): The `isofs_lookup()` on inode is called when the pathname translation routine is diverted from the VFS layer to the isofs layer. `isofs_lookup()` sets the inode operation vector

from the superblock's s_root field of the superblock, and then invokes `isofs_find_entry()` to retrieve the object from the CD-ROM.

On a CD-ROM file system, `inode_operations` map to `isofs_dir_inode_operations`.

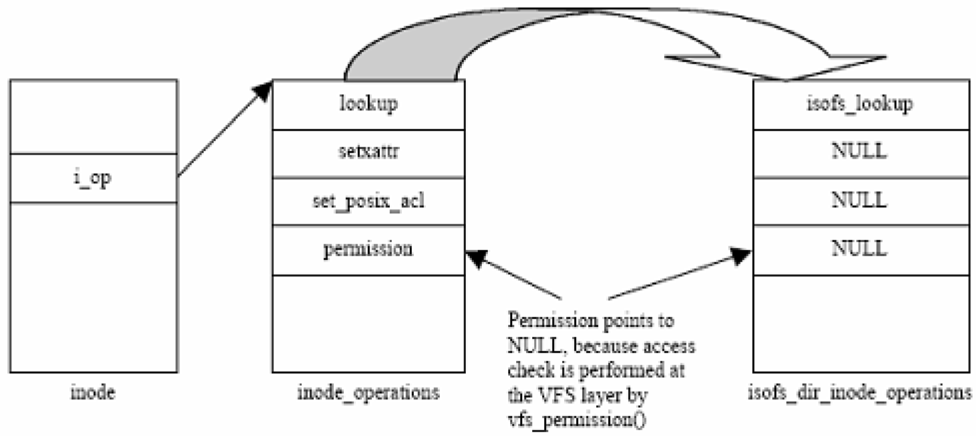


Figure 5-10: File lookup on CD-ROM file system

5.1.3 Pseudo file systems

5.1.3.1 procfs

The proc file system is a special file system that allows system programs and administrators to manipulate the data structures of the kernel. The proc file system is mounted at `/proc`, and provides Virtual File System access to information about current running processes and kernel data structures.

An administrator can change kernel parameters, such as `IP_FORWRDING`, by editing files in `/proc`. For each active process, the kernel creates a directory entry, named after the process ID, in the `/proc` directory. This directory contains pseudo files that can be used to read the status of the process. The Process ID directory is created with a mode of 555 and is owned by the user ID and group ID of the process. Access control is performed by the VFS pathname translation mechanism function `vfs_permission()`, which prevents access by normal users to data of other processes. In addition to `vfs_permission()`, different files in the proc file system define their own access control service functions. These service functions sometimes perform an additional access check that may restrict DAC decisions further.

Root can change permissions for files in `/proc`. The pseudo files within the process directory are only readable for others as far as they provide information similar to the `ps` command. Because files in `/proc` are not real disk-based files with user data, there is no object reuse issue.

5.1.3.2 tmpfs

tmpfs is a memory-based file system that uses virtual memory (VM) resources to store files. tmpfs is designed primarily as a performance enhancement to allow short-lived files to be written and accessed without generating disk or network I/O. tmpfs maximizes file manipulation speed while preserving file semantics.

tmpfs also has dynamic file system size. As a file gets created, the tmpfs file system driver will allocate more VM resources and dynamically increase file system size. In the same way as files get deleted, the file system driver shrinks the size of file system and deallocates VM resources.

Since VM is volatile in nature, tmpfs data is not preserved between reboots. Hence this file system is used to store short-lived temporary files. An administrator is allowed to specify the memory placement policies (the policy itself and the preferred nodes to be allocated) for this file system.

5.1.3.3 sysfs

sysfs is an in-memory file system, which acts as repository for system and device status information, providing a hierarchical view of the system device tree. The system information that is dynamically maintained in the sysfs file system is analogous to the process status information that is dynamically maintained in the proc file system.

sysfs is typically mounted on `/sys`. It is a window into the kernel, and into the data objects that the kernel creates and controls.

sysfs is created on boot, and automatically populated when internal objects are registered with their subsystems. Because of its nature and its design, the hierarchy it creates is a completely accurate representation of the kernel's internals. An administrator can change kernel object parameters by editing files in `/sys`. Access Control is performed by the VFS pathname translation mechanism function `vfs_permission()`, which prevents access by normal users to data belonging to the kernel.

The kernel initially determines permissions for files in `/sys`, but these can be changed. Since files in `/sys` are not real disk-based files, there is no object reuse issue with user data.

5.1.3.4 devpts

The devpts file system is a special file system that provides pseudo terminal support. Pseudo terminals are implemented as character devices. A pair of character device-special files, one corresponding to the master device and the other corresponding to the slave device, represent a pseudo terminal. The slave device provides a terminal interface. Instead of a hardware interface and associated hardware supporting the terminal functions, a process that manipulates the master device of the pseudo terminal implements the interface.

Any data written on the master device is delivered to the slave device, as though it had been received from a hardware interface. Any data written on the slave device can be read from the master device.

In order to acquire a pseudo terminal, a process opens the `/dev/ptmx` master device. The system then makes available to the process number a slave, which can be accessed as `/dev/pts/`*`number`*. An administrator can mount the devpts special file system by providing uid, gid, and mode values on the mount command line. If specified, these values set the owner, group, and mode of the newly created pseudo terminals to the specified values.

In terms of access control, pseudo terminal devices are identical to device special files. Therefore, access control is performed by the VFS pathname translation mechanism function `vfs_permission()`. Because files in `/dev/pts` are not real disk-based files with user data, there is no object reuse issue.

5.1.3.5 rootfs

rootfs is a special file system that the kernel mounts during system initialization. This file system provides an empty directory that serves as an initial mount point, where temporary files can be stored during the boot process. Then, the kernel mounts the real root file system over the empty directory. The rootfs file system allows the kernel to easily change the root file system. Because rootfs uses an empty directory that is replaced by the real root file system before the init process starts, there is no issue of object reuse.

The rootfs is used internally in the kernel when doing root mounting. Because a real file system uses and replaces rootfs before the init process, there is no mechanism to access it.

5.1.3.6 binfmt_misc

binfmt_misc provides the ability to register additional binary formats to the kernel without compiling an additional module or kernel. Therefore, binfmt_misc needs to know magic numbers at the beginning, or the filename extension of the binary.

binfmt_misc works by maintaining a linked list of structs that contain a description of a binary format, including a magic number with size, or the filename extension, offset and mask, and the interpreter name. On request it invokes the given interpreter with the original program as an argument. Because binfmt_misc does not define any default binary-formats, one has to register an additional binary-format. Because files in `/proc/sys/binfmt_misc` are not real disk-based files with user data, there is no object reuse issue.

Refer to `kernel-2.6.18/linux-2.6.18/Documentation/binfmt_misc.txt` for a detailed specification of binfmt_misc.

5.1.3.7 securityfs

Linux Security Modules (LSMs) use securityfs, which is a new virtual file system, avoiding some of the other LSMs to create their own file systems. securityfs must be mounted on `/sys/kernel/security`. To users it appears to be part of the sysfs, but it is a new and distinct one.

5.1.3.8 configfs

configfs is a RAM-based pseudo file system manager of kernel objects. It provides a converse of sysfs functionality through a user space driver kernel object configuration. For additional information about configfs, refer to http://lwn.net/Articles/148973 and http://lwn.net/Articles/149005.

5.1.4 inotify

inotify is a mechanism for watching and communicating file system events to user space. It is an improvement and replacement of the dnotify tool, which had the same purpose. inotify is relevant because different sorts of applications might want or need to know when events such as file changes or creation happen. An example of the use of inotify is with security monitoring applications that must know of, and can benefit from being told about, file system changes.

inotify uses a syscall interface. Applications open a watch file descriptor via the `inotify_init()` call and register watches via the `inotify_add_watch()` call. To add a watch, an application must have DAC read permission on the inode. The access checks are performed by the `vfs_permission()` function. For more information on inotify, see the inotify(7) manage, `Documentation/filesystems/inotify.txt` in the SLES kernel source available.

5.1.5 Discretionary Access Control (DAC)

Previous sections have described how appropriate *_permission() functions are called to perform access checks for non-disk-based and disk-based file systems. Access checks are based on the credentials of the process attempting access, and access rights assigned to the object.

When a file system object is created, the creator becomes the owner of the object. The group ownership (group ID) of the object is set either to the effective group ID of the creator, or to the group ID of the parent directory, depending on the mount options and the mode of the parent directory.

If the file system is mounted with the grpid option, then the object takes the group ID of the directory in which it is created; otherwise, by default, the object takes the effective group ID of the creator, unless the directory has the setgid bit set, in which case the object takes the GID from the parent directory, and also gets the setgid bit set if it is a directory itself. This ownership can be transferred to another user by invoking the chown() system call. The owner and the root user are allowed to define and change access rights for an object.

This following subsection looks at the kernel functions implementing the access checks. The function used depends on the file system; for example, `vfs_permission()` invokes `permission()` which then calls specific *_permission() routines based on the `inode`'s inode operation vector `i_op`. `proc_permission()` is called for files in procfs. `ext3_permission()` is called for the ext3 disk-based file system. If no file system specific *_permission() routine was registered, `generic_permission()` is called to perform the access checks. For some file systems including ext3, the specific *_permission() routine invokes `generic_permission()`. Note that access rights are checked when a file is opened and not on each access. Therefore, modifications to the access rights of file system objects become effective at the next request to open the file.

AppArmor may optionally be loaded. AppArmor additionally restricts which files certain programs may access. AppArmor is controlled by profiles in the `/etc/apparmor.d` directory. When loaded, AppArmor applies additional restrictions to `ping`, `syslogd`, `klogd`, `netstat`, `traceroute`, `lld`, `named`, `identd`, `nscd`, `ntpd`, and `mdnsd`. Additional profiles may be created by an authorized administrator. AppArmor can run without affecting the TOE security functions because AppArmor will only add restrictions, it will not allow what is denied. Whenever DAC denies an operation, AppArmor is not even consulted.

5.1.5.1 Permission bits

`generic_permission()` implements standard UNIX permission bits to provide DAC for file system objects for the procfs, devpts, sysfs, tmpfs, securityfs, binfmt_misc, and ISO 9660 file systems. As noted in Section 5.1.3.5, there is no mechanism for accessing rootfs.

The ext3 file system uses the permission bits for files that do not have associated ACL information. This is implemented in the `generic_permission()` function.

There are three sets of three bits that define access for three categories of users: the owning user, users in the owning group, and other users. The three bits in each set indicate the access permissions granted to each user category: one bit for read (r), one for write (w), and one for execute (x). Note that write access to file systems mounted as read only, such as CDROM, is always rejected. Each subject's access to an object is defined by some combination of these bits:

rwx	indicates read, write, and execute
r-x	indicates read and execute
r--	indicates read
---	indicates null

When a process attempts to reference an object protected only by permission bits, the access is determined as follows:

- Users with an effective user ID of 0 are able to read and write all files, ignoring the permission bits. Users with an effective user ID of zero are also able to execute any file if it is executable for someone.
- If the file system UID equals the object-owning UID, and the owning user permission bits allow the type of access requested, access is granted with no further checks.
- If the file system GID, or any supplementary groups of the process equal an object's owning GID, and the owning group permission bits allow the type of access requested, access is granted with no further checks.

- If the process is neither the owner nor a member of an appropriate group, and the permission bits for world allow the type of access requested, then the subject is permitted access.
- If none of the conditions above are satisfied, and the effective UID of the process is not zero, then the access attempt is denied.

5.1.5.2 Access Control Lists

The ext3 file system supports Access Control Lists (ACLs) that offer more flexibility than the traditional permission bits. An ACL can enforce specific access rights for multiple individual users and groups, not just for the single user and group defined for permission-bit based access control.

The `ext3_check_acl()` function checks if an object has an associated ACL. If it does not have one, the system uses the standard permission bits algorithm as described in the previous section.

If the file system object has an associated ACL, the kernel calls the `posix_acl_permission()` function to enforce POSIX ACLs. ACLs are created, maintained, and used by the kernel. For more detailed information about the POSIX ACLs, refer to the http://acl.bestbits.at and http://wt.xpilot.org/publications/posix.1e sites.

An ACL entry contains the following information:

- A type of tag that specifies the type of the ACL entry.
- A qualifier that specifies an instance a type of an ACL entry.
- A permission set that specifies the discretionary access rights for processes identified by the tag type and qualifier.

5.1.5.2.1 Types of ACL tags

The following types of tags exist:

- `ACL_GROUP`: This type of ACL entry defines access rights for processes whose file system group ID or any supplementary group IDs match the one in the ACL entry qualifier.
- `ACL_GROUP_OBJ`: This type of ACL entry defines access rights for processes whose file system group ID or any supplementary group IDs match the group ID of the group of the file.
- `ACL_MASK`: This type of ACL entry defines the maximum discretionary access rights for a process in the file group class.
- `ACL_OTHER`: This type of ACL entry of this type defines access rights for processes whose attributes do not match any other entry in the ACL.
- `ACL_USER`: An ACL entry of this type defines access rights for processes whose file system user ID matches the ACL entry qualifier.
- `ACL_USER_OBJ`: An ACL entry of this type defines access rights for processes whose file system user ID matches the user ID of the owner of the file.

5.1.5.2.2 ACL qualifier

The qualifier is required for the `ACL_GROUP` and `ACL_USER` ACL types of entries, and contain either the user ID or the group ID for which the access rights are defined.

5.1.5.2.3 ACL permissions

An ACL entry can define separate permissions for read, write, and execute or search.

5.1.5.2.4 Relationship to file permission bits

An ACL contains exactly one entry for each of the `ACL_USER_OBJ`, `ACL_GROUP_OBJ`, and `ACL_OTHER` types of tags, called the required ACL entries. An ACL can have between zero and a defined maximum number of entries of the `ACL_GROUP` and `ACL_USER` types. An ACL that has only the three required ACL entries is called a minimum ACL. ACLs with one or more ACL entries of the `ACL_GROUP` or `ACL_USER` types are called extended ACLs.

The standard UNIX file permission bits as described in the previous section are equivalent to the entries in the minimum ACL. The owner permission bits correspond to the entry of the `ACL_USER_OBJ` type. The entry of the `ACL_GROUP_OBJ` type represents the permission bits of the file group. The entry of the `ACL_OTHER` type represents the permission bits of processes running with an effective user ID and effective group ID or supplementary group ID different from those defined in `ACL_USER_OBJ` and `ACL_GROUP_OBJ` entries.

Minimum ACLs do not need to be stored on disk. The permission information contained in the `inode` is sufficient for the access check. When adding ACL entries to a file system object that did not previously have an explicit ACL, the kernel creates a minimum ACL based on the `inode` attributes, and then adds the new entries to that.

5.1.5.2.5 ACL_MASK

If an ACL contains an `ACL_GROUP` or `ACL_USER` type of entry, then exactly one entry of the `ACL_MASK` type is required in the ACL. Otherwise, the `ACL_MASK` type of entry is optional.

5.1.5.2.6 Default ACLs and ACL inheritance

A default ACL is an additional ACL, which can be associated with a directory. This default ACL has no effect on the access to this directory. Instead, the default ACL is used to initialize the ACL for any file that is created in this directory. When an object is created within a directory, and the ACL is not defined with the function creating the object, the new object inherits the default ACL of its parent directory as its initial ACL. This is implemented by `ext3_create()`, which invokes `ext3_new_inode()`, which in turn invokes `ext3_init_acl()` to set the initial ACL.

5.1.5.2.7 ACL representations and interfaces

ACLs are represented in the kernel as extended attributes. The kernel provides system calls such as `getxattr()`, `setxattr()`, `listxattr()`, and `removexattr()` to create and manipulate extended attributes. User space applications can use these system calls to create and maintain ACLs and other extended attributes. However, ACL applications, instead of directly calling system calls, use library functions provided by the POSIX 1003.1e compliant libacl.so. Inside the kernel, the system calls are implemented using the `getxattr`, `setxattr`, `listxattr`, and `removexattr` inode operations. The kernel provides two additional inode operations, `get_posix_acl()` and `set_posix_acl()`, to allow other parts of the kernel to manipulate ACLs in an internal format that is more efficient to handle than the format used by the inode xattr operations.

In the ext3 disk-based file system, extended attributes are stored in a block of data accessible through the `i_file_acl` field of the `inode`. This extended attribute block stores name-value pairs for all extended attributes associated with the `inode`. These attributes are retrieved and used by appropriate access control functions.

5.1.5.2.8 ACL enforcement

The `ext3_permission()` function uses ACLs to enforce DAC. The algorithm goes through the following steps:

1. Performs checks such as “no write access if read-only file system” and “no write access if the file is immutable.”
2. For ext3 file systems, the kernel calls the `ext3_get_acl()` to get the ACL corresponding to the object. `ext3_get_acl()` calls `ext3_xattr_get()`, which in turn calls `ext3_acl_from_disk()` to retrieve the extended attribute from the disk. If no ACL exists, the kernel follows the permission bits algorithm described previously.
3. For ext3 file systems, the kernel invokes `posix_acl_permission()`. It goes through the following algorithm:

   If the file system user ID of the process matches the user ID of the file object owner, then

   if the `ACL_USER_OBJ` entry contains the requested permissions, access is granted,

   else access is denied.

   else if the file system user ID of the process matches the qualifier of any entry of type `ACL_USER`, then

   if the matching `ACL_USER` entry and the `ACL_MASK` entry contain the requested permissions, access is granted,

   else access is denied.

   else if the file system group ID or any of the supplementary group IDs of the process match the qualifier of the entry of type `ACL_GROUP_OBJ`, or the qualifier of any entry of type `ACL_GROUP`, then

   if the `ACL_MASK` entry and any of the matching `ACL_GROUP_OBJ` or `ACL_GROUP` entries contain all the requested permissions, access is granted,

   else access is denied.

   else if the `ACL_OTHER` entry contains the requested permissions, access is granted.

   else access is denied.

The ACL checking function cycles through each ACL entry to check if the process is authorized to access the object in the attempted mode. Root is always allowed to override any read or write access denials based an ACL entry. Root is allowed to override an attempted execute access only if an execute bit is set for owner, group, or other.

For example, consider a file named `/aclfile`, with mode of 640. The file is owned by root and belongs to the group root. Its default ACL, without the extended POSIX ACL, would be:

```
# owner: root
# group: root
user:: rw-
group::r--
other::---
```

The file is readable and writeable by the root user, and readable by users belonging to the root group. Other users have no access to the file. With POSIX ACLs, a more granular access control can be provided to this file by adding ACLs with the `setfacl` command. For example, the following command allows a user named john read access to this file, even if john does not belong to the root group.

```
#setfacl –m user:john:4,mask::4 /aclfile
```

The ACL on file will look like:

```
# owner: root
# group: root
user:: rw-
user:john:r—
group::r--
mask::r--
other::---
```

The mask field reflects the maximum permission that a user can get. Hence, as per the ACL, even though john is not part of the root group, john is allowed read access to the file `/aclfile`.

5.1.6 Asynchronous I/O

Asynchronous I/O (AIO) enables even a single application thread to overlap I/O operations with other processing, by providing an interface for submitting one or more I/O requests in one system call (`io_submit()`) without waiting for completion, and a separate interface (`io_getevents()`) to reap completed I/O operations associated with a given completion group.

General operation of asynchronous I/O proceeds as follows:

- Process sets up asynchronous I/O context, for files opened with `O_DIRECT`, using `io_setup` system call.
- Process uses `io_submit` system call to submit a set of I/O operations.
- Process checks the completion of I/O operations using `io_getevents`.
- Process destroys the asynchronous I/O context using the `io_destroy` system call.

AIO uses the kernel bottom half mechanism of work queues to perform deferred work of AIO. `io_setup` sets up a work queue named `aio`, to which AIO work is submitted.

Some of the capabilities and features provided by AIO are:

- The ability to submit multiple I/O requests with a single system call.
- The ability to submit an I/O request without waiting for its completion and to overlap the request with other processing.
- Optimization of disk activity by the kernel through combining or reordering the individual requests of a batched I/O variety.
- Better CPU utilization and system throughput by eliminating extra threads and reducing context switches.

5.1.7 I/O scheduler

The I/O scheduler in Linux forms the interface between the generic block layer and the low-level device drivers. The block layer provides functions that are utilized by the file systems and the virtual memory manager to submit I/O requests to block devices. These requests are transformed by the I/O scheduler and made available to the low-level device drivers. The device drivers consume the transformed requests and forward them, by using device-specific protocols, to the actual device controllers that perform the I/O operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the I/O scheduler is considered an important kernel component in the I/O path. SLES includes four I/O scheduler options to optimize system performance.

5.1.7.1 Deadline I/O scheduler

The deadline I/O scheduler available in the Linux 2.6 kernel incorporates a per-request expiration-based approach, and operates on five I/O queues. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance, while simultaneously ensuring that no I/O request is being starved. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to read than write requests.

As stated earlier, the deadline I/O scheduler maintains five I/O queues. During the enqueue phase, each I/O request gets associated with a deadline, and is inserted into I/O queues that are either organized by starting block (a sorted list) or by the deadline factor (a first-in-first-out [FIFO]) list. The scheduler incorporates separate sort and FIFO lists for read and write requests, respectively. The fifth I/O queue contains the requests that are to be handed off to the device driver. During a dequeue operation, in the case that the dispatch queue is empty, requests are moved from one of the four I/O lists (sort or FIFO) in batches. The next step consists of passing the head request on the dispatch queue to the device driver.

The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each read request is processed by its effective deadline without actually starving the queued-up write requests. In this design, the goal of economizing on the disk seek time is accomplished by moving a larger batch of requests from the sort list, which is sector sorted, and balancing it with a controlled number of requests from the FIFO list.

5.1.7.2 Anticipatory I/O scheduler

The design of the anticipatory I/O scheduler attempts to reduce the per-thread read response time. It introduces a controlled delay component into the dispatching equation. The delay is invoked on any new request to the device driver, thereby allowing a thread that just finished its I/O request to submit a new request.

Implementation of the anticipatory I/O scheduler is similar to, and may be considered as, an extension to the deadline scheduler. In general, the scheduler follows the basic idea that if the disk drive just operated on a read request, the assumption can be made that there is another read request in the pipeline, and hence it is worthwhile to wait. The I/O scheduler starts a timer, and at this point, there are no more I/O requests passed down to the device driver. If a close read request arrives during the wait time, it is serviced immediately, and in the process, the actual distance that the kernel considers as close grows as time passes, which is the adaptive part of the heuristic. Eventually the close requests dry out, causing the scheduler to submit some of the write requests, converging back to what is considered a normal I/O request dispatching scenario.

5.1.7.3 Completely Fair Queuing scheduler

The Completely Fair Queuing (CFQ) I/O scheduler can be considered an extension to the Stochastic Fair Queuing (SFQ) scheduler implementation. The focus of both implementations is on the concept of fair allocation of I/O bandwidth among all the initiators of I/O requests. The actual implementation utilizes n (normally 64) internal I/O queues, as well as a single I/O dispatch queue.

During an enqueue operation, the PID of the currently running process (the actual I/O request producer) is utilized to select one of the internal queues, which is normally hash-based, and, hence, the request is inserted into one of the queues in FIFO order. During dequeue, it calls for a round-robin based scan through the non-empty I/O queues, and selects requests from the head of the queues. To avoid encountering too many seek operations, an entire round of requests is first collected, sorted, and ultimately merged into the dispatch queue.

Next, the head request in the dispatch queue is passed to the actual device driver. The CFQ I/O scheduler implements time sharing, in which the processes possess time slices during which they can dispatch I/O requests. This capability makes it behaves similarly to the Anticipatory I/O scheduler. I/O priorities are also considered for the processes, which are derived from their CPU priority.

5.1.7.4 Noop I/O scheduler

The noop I/O scheduler can be considered as a rather minimal I/O scheduler that performs, as well as provides, basic merging and sorting functionalities. The main usage of the noop scheduler revolves around non-disk-based block devices, such as memory devices, as well as specialized software or hardware environments that incorporate their own I/O scheduling and large caching functionality, thus requiring only minimal assistance from the kernel.

5.1.8 I/O interrupts

The Linux kernel supports concurrent execution of multiple tasks. Each active task gets a portion of the CPU time to advance its execution. Apart from this, the CPU also has to respond to address space violations, page faults, synchronous signals from the CPU control unit, and asynchronous signals from devices such as a keyboard or a network card. This section describes how the Linux kernel handles these asynchronous interrupts generated by I/O devices.

Various I/O devices, such as the keyboard, communicate with the CPU regarding events occurring in the device, such as a key typed at the keyboard, by sending special electrical signals to the CPU. The CPU receives the signal and communicates it to the kernel for processing. Depending on the signal, the kernel executes an appropriate interrupt handler to process the event.

Responsiveness of the system can be increased by promptly handling the interrupts. However, depending on the type of the interrupt, not all actions associated with handling an interrupt must be executed immediately. Therefore, an interrupt handler can be thought to consist of two sets of operations.

The first set, which is called *the top half*, consists of operations that must be executed immediately. The second set, which is called *the bottom half*, consists of operations that can be deferred. The top half usually includes the most critical tasks, such as acknowledging the signal. The Linux kernel provides three mechanisms for implementing a bottom half of an interrupt handler. They are *softirqs*, *tasklets*, and *work queues*.

5.1.8.1 Top halves

Top halves perform critical parts of interrupt-related tasks such as acknowledging interrupts to the PIC, reprogramming the PIC or device controller, and updating data structures accessed by both device and processor.

5.1.8.2 Bottom halves

Bottom halves perform interrupt-related tasks that were not performed by the top half of the interrupt handler. That is, bottom halves perform the work that was deferred by the top halves because it was not absolutely necessary to perform it in the top half.

5.1.8.3 Softirqs

Softirqs are statically linked (defined at compile time) bottom halves that execute in the interrupt context. Many softirqs can always be executed concurrently on several CPUs even if they are of same type.

5.1.8.4 Tasklets

Tasklets are dynamically linked and built on top of softirq mechanisms. Tasklets differ from softirqs in that a tasklet is always serialized with respect to itself. In other words, a tasklet cannot be executed by two CPUs at the same time. However, different tasklets can be executed concurrently on several CPUs.

5.1.8.5 Work queue

The work queue mechanism was introduced in the 2.6 Linux kernel. Work queues execute in process context, as opposed to the interrupt context of softirqs and tasklets. Work queues defer work to kernel threads, which are schedulable, and can therefore sleep. Thus, work queues provide an interface to create kernel threads to handle work queued from other operations. The work queue infrastructure allows a device driver to create its own kernel thread or use generic worker threads, one per processor, provided by the kernel.

5.1.9 Processor interrupts

A symmetrical multiprocessing (SMP) system sets slightly different requirements to interrupt handling by hardware and software than an ordinary uniprocessing (UP) system. Distributed handling of hardware interrupts has to be implemented to take advantage of parallelism, and an efficient mechanism for communicating between CPUs must be provided.

Inter-processor interrupts (IPIs) are used to exchange messages between CPUs in SMP system. The following group of functions helps in issuing IPIs:

- `send_IPI_all()` Sends an IPI to all CPUs (including the sender)
- `send_IPI_allbutself()` Sends an IPI to all CPUs except the sender
- `send_IPI_self()` Sends an IPI to the sender CPU
- `send_IPI_single()` Sends an IPI to a single, specified CPU

On a multiprocessor system, Linux defines the following five types of IPIs:

- CALL_FUNCTION_VECTOR(vector 0xfb) Used to call functions with a given argument on other CPUs like flush `tlb_all_ipi()` and `stop_this_cpu()`. Handler: `smp_call_function_interrupt()`.
- RESCHEDULE_VECTOR(0xfc) Used at least when the best CPU for the woken up task is not this CPU. Handler: `smp_reschedule_interrupt()`.
- INVALIDATE_TLB_VECTOR(VECTOR 0xfd) Used when the TLBs of the other CPU need to be invalidated. Handler: `smp_invalidate_interrupt()`.
- ERROR_APIC_VECTOR(vector 0xfe) This interrupt should never occur.
- SPURIOUS_APIC_VECTOR(vector 0xff) This interrupt should never occur.

5.1.10 Machine check

A machine check exception is an imprecise non-recoverable exception, which means that the CPU does not guarantee it will give a coherent set of register values after the exception occurs. A machine check exception occurs when something goes wrong inside the CPU, such as cosmic rays causing bits to randomly flip, or the CPU overheating causing electrons to drift.

5.2 Process control and management

A process is an instance of a program in execution. Process management consists of creating, manipulating, and terminating a process. Process management is handled by the process management subsystems of the kernel. The kernel interacts with the memory subsystem, the network subsystem, the file and I/O subsystem, and the inter-process communication (IPC) subsystem.

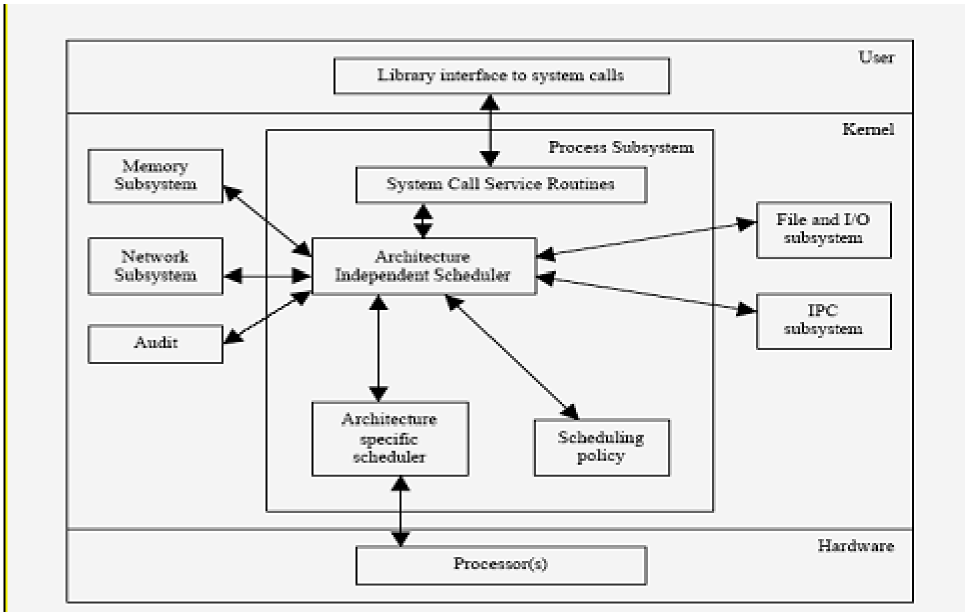


Figure 5-11: Process subsystem and its interaction with other subsystems

The kernel treats a process as a subject. A subject is an active entity that can access and manipulate data and data repositories, which are objects. System resources, such as CPU time and memory, are allocated to objects. The kernel manages a process through a number of data structures. These data structures are created, manipulated, and destroyed to give processes viability.

This section briefly describes how a process is given credentials that are used in access mediation, and how the credentials are affected by process and kernel actions during the life cycle of the process.

This section is divided into four subsections. Data Structures lists important structures that are used to implement processes and highlight security relevant credentials fields. Process Creation and Destruction describes creation, destruction, and maintenance of a process with emphasis on how security-relevant credentials are affected by state transitions. Process Switch describes how the kernel switches the current process that is executing on the processor, with emphasis on mechanisms that ensure a clean switch (that is, ensuring that the latest process executing is not using any resources from the switched out process). Kernel Threads describes special-purpose subjects that are created to perform critical system tasks.

5.2.1 Data structures

The SLES kernel provides two abstractions for subject constructs: a regular process and a lightweight process. A lightweight process differs from a regular process in its ability to share some resources, such as address space and open files. With respect to security relevance, if differences exist between regular processes and lightweight processes, those differences are highlighted. Otherwise, both regular and lightweight processes are simply referred to as processes for better readability.

The SLES kernel maintains information about each process in a `task_struct` process type of descriptor. Each process descriptor contains information such as run-state of process, address space, list of open files, process priority, which files the process is allowed to access, and security relevant credentials fields including the following:

- uid and gid, which describe the user ID and group ID of a process.
- euid and egid, which describe the effective user ID and effective group ID of a process.
- fsuid and fsgid, which describe the file system user ID and file system group ID of a process.
- suid and sgid, which describe the saved user ID and saved group ID of a process.
- groups, which lists the groups to which the process belongs.
- state, which describes the run state of the process.
- pid, which is the process identifier used by the kernel and user processes for identification.
- security, which points to the information relating to the process domain and other attributes used and managed by AppArmor.

The credentials are used every time a process tries to access a file or IPC objects. Process credentials, along with the access control data and ownership of the object, determine if access is allowed.

Refer to `include/linux/sched.h` for information about other `task_struct` fields.

Figure 5-12 schematically shows the `task_struct` structure with fields relevant for access control.

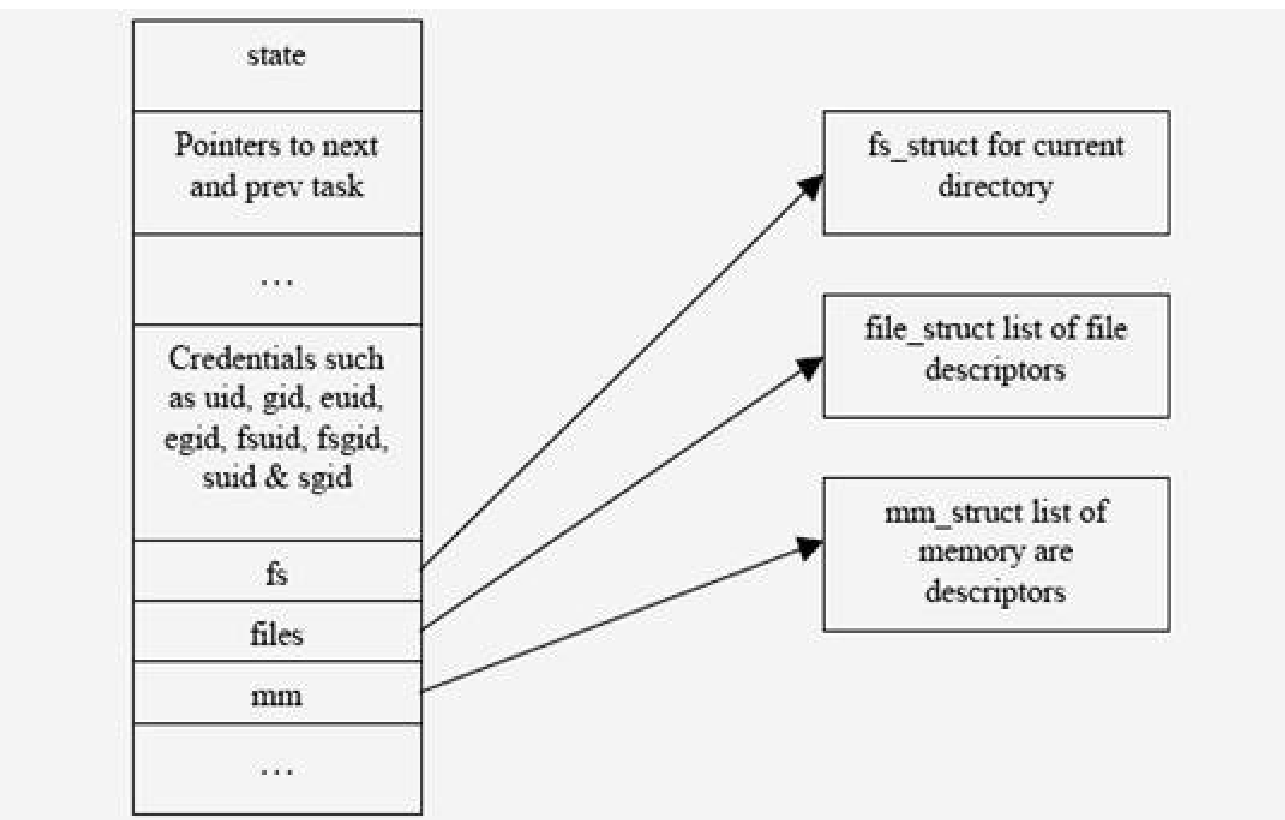


Figure 5-12: The task structure

The kernel maintains a circular doubly-linked list of all existing process descriptors. The head of the list is the `init_task` descriptor referenced by the first element of the task array. The `init_task` descriptor belongs to process 0 or the swapper, the ancestor of all processes.

5.2.2 Process creation and destruction

The SLES kernel provides these system calls for creating a new process: `clone()`, `fork()`, and `vfork()`. When a new process is created, resources owned by the parent process are duplicated in the child process. Because this duplication is done using memory regions and demand paging, the object reuse requirement is satisfied.

The `vfork()` system call differs from `fork()` by sharing the address space of its parent. To prevent the parent from overwriting data needed by the child, the execution of the parent is blocked until the child exits or executes a new program. Lightweight processes are created using the `clone()` system call, which allows both the parent and the child to share many per-process kernel data structures such as paging tables, open file tables, and signal dispositions.

5.2.2.1 Control of child processes

The child process inherits the parent's security-relevant credentials, including uid, euid, gid, and egid. Because these credentials are used for access control decisions, the child is given the same level of access to objects as the parent. The credentials of a child changes when it starts executing a new program or issues suitable system calls, which are listed as follows:

5.2.2.2 DAC controls

5.2.2.2.1 setuid()and setgid()

These set the effective user and group ID of the current process. If the effective user ID of the caller is root, then the real and saved user and group IDs are also set.

5.2.2.2.2 seteuid()and setegid()

These set the effective user and group ID of the current process. Normal user processes may only set the effective user and group ID to the real user and group ID, the effective user and group ID, or the saved user and group ID.

5.2.2.2.3 setreuid()and setregid()

These set the real and effective user and group IDs of the current process. Normal users may only set the real user and group ID to the real user and group ID or the effective user and group ID, and can only set the effective user and group ID to the real user and group ID, the effective user and group ID or the saved user and group ID. If the real user and group ID is set or the effective user and group ID is set to a value not equal to the previous real user and group ID, the saved user and group ID is set to the new effective user and group ID.

5.2.2.2.4 setresuid()and setresgid()

These set the real user and group ID, the effective user and group ID, and the saved set-user and group ID of the current process. Normal user processes (that is, processes with real, effective, and saved user IDs that are nonzero) may change the real, effective, and saved user and group IDs to either the current uid and gid, the current effective uid and gid, or the current saved uid and gid. An administrator can set the real, effective, and saved user and group ID to an arbitrary value.

5.2.2.3 execve()

This invokes the `exec_mmap()` function to release the memory descriptor, all memory regions, and all page frames assigned to the process, and to clean up the Page Tables of a process. The `execve()` function invokes the `do_mmap()` function twice, first to create a new memory region that maps the text segment of the executable, and then to create a new memory region that maps the data segment of the executable file. The object reuse requirement is satisfied because memory region allocation follows the demand paging technique described in Section 5.5.

`execve()` can also alter the credentials of the process if the setuid bit of the executable file is set. If the setuid bit is set, the current euid and fsuid of the process are set to the identifier of the owner of the file. This change of credentials affects process permissions for the DAC policy.

5.2.2.4 do_exit()

Process termination is handled in the kernel by the `do_exit()` function. The `do_exit()` function removes most references to the terminating process from the kernel data structures and releases resources, such as memory, open files, and semaphores held by the process.

5.2.3 Process switch

To control the execution of multiple processes, the SLES kernel suspends the execution of the process currently running on the CPU and resumes the execution of some other process previously suspended. In performing a process switch, the SLES kernel ensures that each register is loaded with the value it had when the process was suspended. The set of data that must be loaded into registers is called the hardware context, which is part of the larger process execution context. Part of the hardware context is contained in the task structure of a process; the rest is saved in the kernel mode stack of a process, which allows for the separation needed for a clean switch. In a three-step process, the switch is performed by:

1. installation of a new address space
2. switching the Kernel Mode Stack
3. switching the hardware context

5.2.4 Kernel threads

The SLES kernel delegates certain critical system tasks, such as flushing disk caches, swapping out unused page frames, and servicing network connections, to kernel threads. Because kernel threads execute only in kernel mode, they do not have to worry about credentials. Kernel threads satisfy the object reuse requirement by allocating memory from the kernel memory pool, as described in the kernel memory management section of this document.

5.2.5 Scheduling

Scheduling is one of the features that is highly improved in the SLES 2.6 kernel over the 2.4 kernel. It uses a new scheduler algorithm, called the O (1) algorithm, that provides greatly increased scheduling scalability. The O (1) algorithm achieves this by taking care that the time taken to choose a process for placing into execution is constant, regardless of the number of processes. The new scheduler scales well, regardless of process count or processor count, and imposes a low overhead on the system.

In the Linux 2.6 scheduler, time to select the best task and get it on a processor is constant, regardless of the load on the system or the number of CPUs for which it is scheduling.

Instead of one queue for the whole system, one active queue is created for each of the 140 possible priorities for each CPU. As tasks gain or lose priority, they are dropped into the appropriate queue on the processor on which they last ran. Now it is easy for a processor to find the highest priority task. As tasks complete their time slices, they go into a set of 140 parallel queues, named the expired queues, per processor. When the active queue is empty, a simple pointer assignment can cause the expired queue to become the active queue again, making turnaround quite efficient.

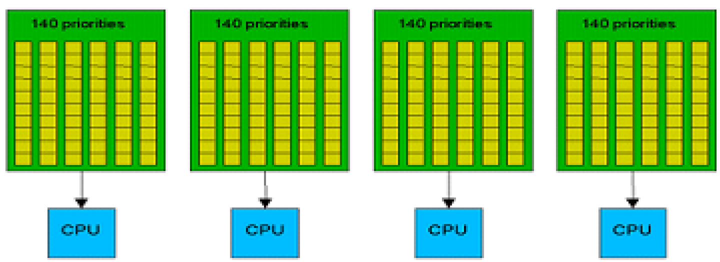


Figure 5-13: O(1) scheduling

For more information about O(1) scheduling, refer to *Linux Kernel Development - A Practical guide to the design and implementation of the Linux Kernel*, Chapter 3, by Robert Love, or “Towards Linux 2.6: A look into the workings of the next new kernel” by Anand K. Santhanam at http://www-106.ibm.com/developerworks/linux/library/l-inside.html#h1.

The SLES kernel also provides support for hyperthreaded CPUs that improves hyperthreaded CPU performance. Hyperthreading is a technique in which a single physical processor masquerades at the hardware level as two or more processors. It enables multi-threaded server software applications to execute threads in parallel within each individual server processor, thereby improving transaction rates and response times.

Hyperthreading scheduler

This section describes scheduler support for hyperthreaded CPUs. Hyperthreading support ensures that the scheduler can distinguish between physical CPUs and logical, or hyperthreaded, CPUs. Scheduler compute queues are implemented for each physical CPU, rather than each logical CPU as was previously the case. This results in processes being evenly spread across physical CPUs, thereby maximizing utilization of resources such as CPU caches and instruction buffers.

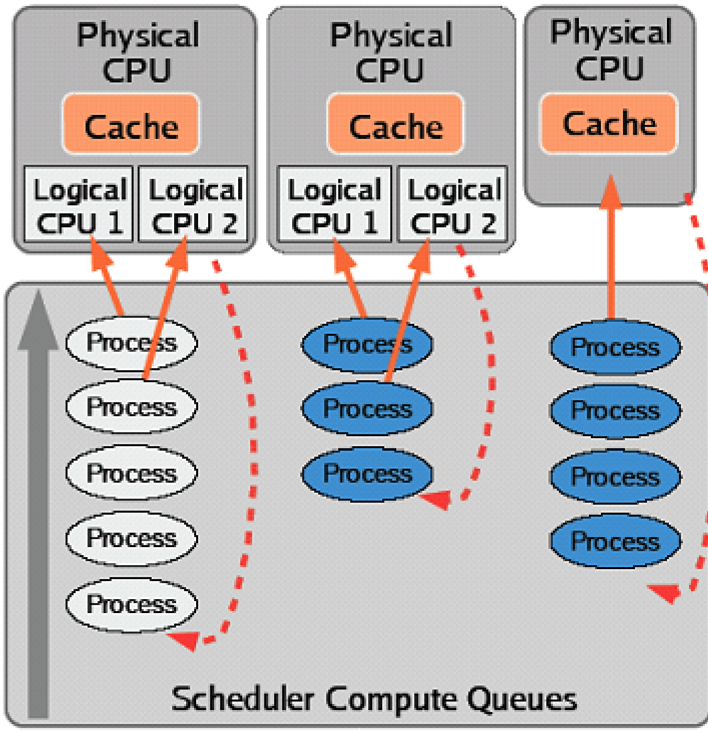


Figure 5-14: Hyperthreaded scheduling

For more information about hyperthreading, refer to http://www.intel.com/technology/hyperthread/.

5.2.6 Kernel preemption

The kernel preemption feature has been implemented in the Linux 2.6 kernel. This should significantly lower latency times for user-interactive applications, multimedia applications, and the like. This feature is especially good for real-time systems and embedded devices. In previous versions of the kernel it was not possible to preempt a task executing in kernel mode, including user tasks that had entered into kernel mode via system calls, until or unless the task voluntarily relinquished the CPU.

Because the kernel is preemptible, a kernel task can be preempted so that some important user applications can continue to run. The main advantage of this is that there will be a big boost to user interactiveness of the system, and the user will feel that things are happening at a faster pace for a key stroke or mouse click.

Of course, not all sections of the kernel code can be preempted. Certain critical sections of the kernel code are locked against preemption. Locking should ensure that both per-CPU data structures and state are always protected against preemption.

The following code snippet demonstrates the per-CPU data structure problem, in an SMP system:

```
int arr[NR_CPUS];
arr[smp_processor_id()] = i;
/* kernel preemption could happen here */
j = arr[smp_processor_id()];
/* i and j are not equal as smp_processor_id() may not be the same */
```

In this situation, if kernel preemption had happened at the specified point, the task would have been assigned to some other processor upon re-schedule, in which case `smp_processor_id()` would have returned a different value. This situation should be prevented by locking.

FPU mode is another case where the state of the CPU should be protected from preemption. When the kernel is executing floating point instructions, the FPU state is not saved. If preemption happens here, then upon reschedule, the FPU state is completely different from what was there before preemption. So, FPU code must always be locked against kernel preemption.

Locking can be done by disabling preemption for the critical section and re-enabling it afterwards. The Linux 2.6 kernel has provided the following #defines to disable and enable preemption:

- `preempt_enable()` -- decrements the preempt counter
- `preempt_disable()` -- increments the preempt counter
- `get_cpu()` -- calls `preempt_disable()` followed by a call to `smp_processor_id()`
- `put_cpu()` -- re-enables preemption

Using these #defines we could rewrite the above code as

```
int cpu, arr[NR_CPUS];
arr[get_cpu()] = i;  /* disable preemption */
j = arr[smp_processor_id()];
/* do some critical stuff here */
put_cpu();    /* re-enable preemption */
```

Note that `preempt_disable()` and `preempt_enable()` calls are nested. That is, `preempt_disable()` can be called **n** number of times, and preemption will only be re-enabled when the **n**th `preempt_enable()` is encountered.

Preemption is implicitly disabled if any spin locks are held. For instance, a call to `spin_lock_irqsave()` implicitly prevents preemption by calling `preempt_disable()`; a call to `spin_unlock_irqrestore()` re-enables preemption by calling `preempt_enable()`.

5.3 Inter-process communication

The SLES kernel provides a number of Inter-process communication (IPC) mechanisms that allow processes to exchange arbitrary amounts of data and synchronize execution. The IPC mechanisms include unnamed pipes, named pipes (FIFOs), the System V IPC mechanisms (consisting of message queues, semaphores and shared memory regions), signals, and sockets.

This section describes the general functionality and implementation of each IPC mechanism and focuses on DAC and object reuse handling.

5.3.1 Pipes

Pipes allow the transfer of data in a FIFO manner. The `pipe()` system call creates unnamed pipes. Unnamed pipes are only accessible to the creating process and its descendants through file descriptors. Once a pipe is created, a process may use the `read()` and `write()` VFS system calls to access it.

In order to allow access from the VFS layer, the kernel creates an `inode` object and two file objects for each pipe. One file object is used for reading (reader) and the other for writing (writer). It is the responsibility of the process to use the appropriate file descriptor for reading and writing. Processes access unnamed pipes through their VFS file descriptors. Hence, access control is performed at the VFS layer in the same manner as for regular files, as described in Sections 5.1.5.

The internal implementation of pipes has changed with the 2.6 kernel. Before, a pipe used a single page to buffer data between the file object reader and writer. For a process writing more than a single page, it became blocked until the file object reader consumed the amount of data necessary to allow the rest to be fit in the buffer. In the new implementation, known as circular pipes, a circular buffer is used.

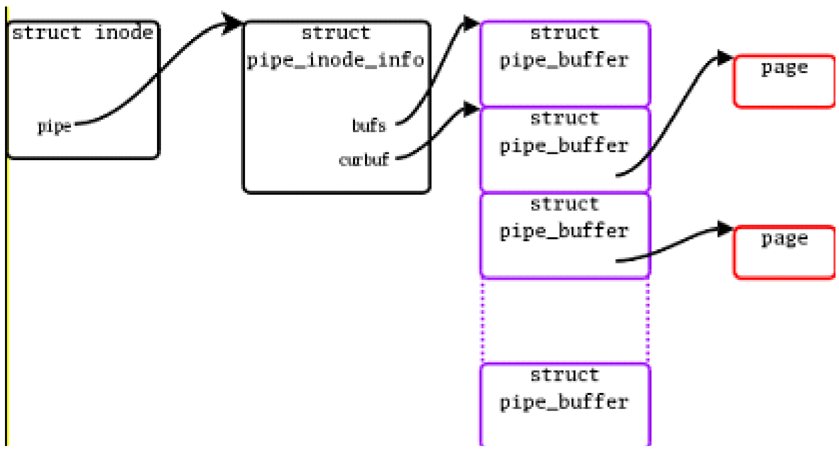


Figure 5-15: Pipes Implementation

In a simple scenario, a `curbuf` pointer indicates the first buffer that contains data in the array, and `nrbufs` indicates the number of buffers that contain data. The page structures are allocated and used as necessary. In order to serialize access, the pipe semaphore is used, since file object writers and readers are able to manipulate `nrbufs`. Length and offset fields compose the pipe buffer structure in order for each entry in the circular buffer to be able to contain less than a full page of data.

This circular implementation improves pipe bandwidth from 30% to 90%, with a small increase in latency because pages are allocated when data passes through the pipe. The better results in performance are attributable to the large buffering, since file object readers and writers block less often when passing data through the pipe.

This new functionality implemented in circular pipes is intended to become a general mechanism for transmitting data streams through the kernel.

5.3.1.1 Data structures and algorithms

The `inode` object refers to a pipe with its `i_pipe` field, which points to a `pipe_inode_info` structure. The `pipe()` system call invokes `do_pipe()` to create a pipe. The `read()` and `write()` operations performed on the appropriate pipe file descriptors invoke, through the file operations vector `f_op` of the file object, the `pipe_read()` and `pipe_write()` routines, respectively.

`pipe_inode_info`: Contains generic state information about the pipe with fields such as `base` (which points to the kernel buffer), `len` (which represents the number of bytes written into the buffer and yet to be read), `wait` (which represents the wait queue), and `start` (which points to the read position in the kernel buffer).

`do_pipe()`: Invoked through the `pipe()` system call, `do_pipe()` creates a pipe that performs the following actions:

1. Allocates and initializes an `inode`.
2. Allocates a `pipe_inode_info` structure and stores its address in the `i_pipe` field of the inode.
3. Allocates a page-frame buffer for the pipe buffer using `__get_free_page()`, which in turn `invokes alloc_pages()` for the page allocation. Even though the allocated page is not explicitly zeroed-out, because of the way `pipe_read()` and `pipe_write()` are written, it is not possible to read beyond what the write channel writes. Therefore, there are no object reuse issues.

`pipe_read()`: Invoked through the `read()` system call, `pipe_read()` reads the pipe buffer that the base field of the `pipe_info` structure points to.

`pipe_write()`: Invoked through the `write()` system call, `pipe_write()` writes in the pipe buffer pointed to by the base field of the `pipe_info` structure.

Because unnamed pipes can only be used by a process and its descendants that share file descriptors, there are no DAC issues.

5.3.2 First-In First-Out Named pipes

A First-In First-Out (FIFO) named pipe is very similar to the unnamed pipe described in Section 5.3.1. Unlike the unnamed pipe, a FIFO has an entry in the disk-based file system. A large portion of the internal implementation of a FIFO pipe is identical to that of the unnamed pipe. Both use the same data structure, `pipe_inode_info`, and the `pipe_read()` and `pipe_write()` routines. The only differences are that FIFOs are visible on the system directory tree and are a bi-directional communication channel. Access control on named pipes is also performed using interaction of processes with file descriptors in a similar manner as for those of access control on regular files, as explained in Section 5.1.5 (DAC).

5.3.2.1 FIFO creation

FIFO exists as a persistent directory entry on the system directory tree. A FIFO is created with the VFS `mknod()` system call, as follows:

1. The `mknod()` call uses the path name translation routines to obtain the `dentry` object of the directory where the FIFO is to be created, and then invokes `vfs_mknod()`.
2. The `vfs_mknod()` call crosses over to the disk-based file system layer by invoking the disk-based file system version of mknod (`ext3_mknod()`) through the `inode` operations vector `i_op`.
3. A special FIFO `inode` is created and initialized. The file operation vector of the `inode` is set to `def_fifo_fops` by a call to function `init_special_inode()`. The only valid file operation in `def_fifo_fops is fifo_open()`.

The creator of the FIFO becomes its owner. This ownership can be transferred to another user using the `chown()` system call. The owner and root user are allowed to define and modify access rights associated with the FIFO.

The inode allocation routine of the disk-based file system does the allocation and initialization of the inode object; thus, object reuse is handled by the disk-based file system.

5.3.2.2 FIFO open

A call to the open() VFS system call performs the same operation as it does for device special files. Regular DACs when the FIFO inode is read are identical to access checks performed for other file system objects, such as files and directories. If the process is allowed to access the FIFO inode, the kernel proceeds by invoking init_special_inode(), because a FIFO on disk appears as a special file. The init_special_inode() system call sets the file operation vector i_fop of the inode to def_fifo_fops. The only valid function in def_fifo_fops is the fifo_open() function. fifo_open() appropriately calls the pipe_read() or pipe_write() functions, depending on the access type. Access control is performed by the disk-based file system.

5.3.3 System V IPC

The System V IPC consists of message queues, semaphores, and shared memory regions. Message queues allow formatted data streams that are sent between processes. Semaphores allow processes to synchronize execution. Shared memory segments allow multiple processes to share a portion of their virtual address space.

This section describes data structures and algorithms used by the SLES kernel to implement the System V IPC. This section also focuses on the implementation of the enforcement of DAC and the handling of object reuse by the allocation algorithms.

- The IPC mechanisms share the following common properties:
- Each mechanism is represented by a table in kernel memory whose entries define an instance of the mechanism.
- Each table entry contains a numeric key, which is used to reference a specific instance of the mechanism.
- Each table entry has an ownership designation and access permissions structure associated with it. The creator of an IPC object becomes its owner. This ownership can be transferred by the control system call of the IPC mechanism. The owner and root user are allowed to define and modify access permissions to the IPC object. Credentials of the process attempting access, ownership designation, and access permissions are used for enforcing DAC. The root user is allowed to override DAC setup through access permissions.
- Each table entry has a pointer to an ipc_security_struct type, which is not used by the SLES kernel.
- Each table entry includes status information such as time of last access or update.
- Each mechanism has a control system call to query and set status information, and to remove an instance of a mechanism.

5.3.3.1 Common data structures

The following list describes security-relevant common data structures that are used by all three IPC mechanisms:

- ipc_ids: The ipc_ids data structure fields, such as size, which indicates the maximum number of allocatable IPC resources; in_use, which holds the number of allocated IPC resources; and, entries, which points to the array of IPC resource descriptors.

- `ipc_id`: The `ipc_id` data structure describes the security credentials of an IPC resource with the `p` field, which is a pointer to the credential structure of the resource.
- `kern_ipc_perm`: The `kern_ipc_perm` data structure is a credential structure for an IPC resource with fields such as key, uid, gid, cuid, cgid, mode, seq, and security. uid and cuid represent the owner and creator user ID. gid and cgid represent the owner and creator group ID. The mode field represents the permission bit mask and the seq field identifies the slot usage sequence number. The security field is a pointer to a structure that is not used by the SLES kernel.

5.3.3.2 Common functions

Common security-relevant functions are `ipc_alloc()` and `ipcperms()`.

5.3.3.2.1 ipc_alloc()

The `ipc_alloc()` function is invoked from the initialization functions of all three IPC resources to allocate storage space for respective arrays of IPC resource descriptors of the IPC resource. The `ipc_ids` data structure field entries point to the IPC resource descriptors. Depending on the size, computed from the maximum number of IPC resources, `ipc_alloc()` invokes either `kmalloc()` with the `GFP_KERNEL` flag, or `vmalloc()`. There are no object reuse issues, because in both cases the memory allocated is in the kernel buffer and the kernel uses the memory for the kernel's internal purposes.

5.3.3.2.2 ipcperms()

The `ipcperms()` function is called when a process attempts to access an IPC resource. `ipcperms()` enforces the DAC policy. Discretionary access to the IPC resource is granted based on the same logic as that of regular files, using the owner, group, and access mode of the object. The only difference is that the owner and creator of the IPC resource are treated equivalently, and the execute permission flag is not used.

5.3.3.3 Message queues

Important data structures for message queues are `msg_queue`, which describes the structure of a message queue, and `msg_msg`, which describes the structure of the message. Important functions for message queues are `msgget()`, `msgsnd()`, `msgrcv()`, and `msgctl()`. Once marked for deletion, no further operation on a message queue is possible.

5.3.3.3.1 msg_queue

This structure describes the format of a message queue with fields such as `q_perm`, which points to the `kern_ipc_perm` data structure; `q_stime`, which contains the time of the last `msgsnd()`; `q_qcbytes`, which contains the number of bytes in `queue q`, and, `qnum`, which contains the number of messages in a queue.

5.3.3.3.2 msg_msg

This structure describes the format of a message with fields such as `m_type`, which specifies the message type; `m_ts`, which specifies message text size; `m_list`, which points to the message list; and, `next`, which points to `msg_msgseg` corresponding to the next page frame containing the message.

5.3.3.3.3 msgget()

This function is invoked to create a new message queue, or to get a descriptor of an existing queue based on a key. The newly created credentials of the message queue are initialized from the credentials of the creating process.

5.3.3.3.4 msgsnd()

This function is invoked to send a message to a message queue. DAC is performed by invoking the `ipcperms()` function. A message is copied from the user buffer into the newly allocated `msg_msg` structure. Page frames are allocated in the buffer space of the kernel using `kmalloc()` and the `GFP_KERNEL` flag. Thus, no special object reuse handling is required.

5.3.3.3.5 msgrcv()

This function is invoked to receive a message from a message queue. DAC is performed by invoking the `ipcperms()` function.

5.3.3.3.6 msgctl()

This function is invoked to set attributes of, query status of, or delete a message queue. Message queues are not deleted until the process waiting for the message has received it. DAC is performed by invoking the `ipcperms()` function.

5.3.3.4 Semaphores

Semaphores allow processes to synchronize execution by performing a set of operations atomically on themselves. An important data structure implementing semaphores in the kernel is `sem_array`, which describes the structure of the semaphore. Important functions are `semget()`, `semop()`, and `semctl()`. Once marked for deletion, no further operation on a semaphore is possible.

5.3.3.4.1 sem_array

Describes the structure and state information for a semaphore object. `sem_array` contains fields including as `sem_perm`, the `kern_ipc_perm` data structure; `sem_base`, which is a pointer to the first semaphore; and, `sem_pending`, which is a pointer to pending operations.

5.3.3.4.2 semget()

A function that is invoked to create a new semaphore or to get a descriptor of an existing semaphore based on a key. The newly created semaphore's credentials are initialized from the creating process's credentials. The newly allocated semaphores are explicitly initialized to zero by a call to `memset()`.

5.3.3.4.3 semop()

This function is invoked to perform atomic operations on semaphores. DAC is performed by invoking the `ipcperms()` function.

5.3.3.4.4 semctl()

A function that is invoked to set attributes, query status, or delete a semaphore. A semaphore is not deleted until the process waiting for a semaphore has received it. DAC is performed by invoking the `ipcperms()` function.

5.3.3.5 Shared memory regions

Shared memory regions allow two or more processes to access common data by placing the processes in an IPC shared memory region. Each process that wants to access the data in an IPC shared memory region adds to its address space a new memory region, which maps the page frames associated with the IPC shared memory region. Shared memory regions are implemented in the kernel using the `shmid_kernel` data structure, and the `shmat()`, `shmdt()`, `shmget()` and `shmctl()` functions.

5.3.3.5.1 shmid_kernel

Describes the structure and state information of a shared memory region with fields including `shm_perm`, which stores credentials in the `kern_ipc_perm` data structure; `shm_file`, which is the special file of the segment; `shm_nattach`, which holds the number of current attaches; and, `shm_segsz`, which is set to the size of the segment.

5.3.3.5.2 shmget()

A function that is invoked to create a new shared memory region or to get a descriptor of an existing shared memory region based on a key. A newly created shared memory segment's credentials are initialized from the creating process's credentials. `shmget()` invokes `newseg()` to initialize the shared memory region. `newseg()` invokes `shmem_file_setup()` to set up the `shm_file` field of the shared memory region. `shmem_file_setup()` calls `get_empty_filp()` to allocate a new file pointer, and explicitly zeroes it out to ensure that the file pointer does not contain any residual data.

5.3.3.5.3 shmat()

A process invokes `shmat()` to attach a shared memory region to its address space. DAC is are performed by invoking the `ipcperms()` function.

The pages are added to a process with the demand paging technique described in Section 5.5.2.5.6. Hence, the pages are dummy pages. The function adds a new memory region to the address space of the process, but actual memory pages are not allocated until the process tries to access the new address for a write operation. When the memory pages are allocated, they are explicitly zeroed out, satisfying the object reuse requirement, as described in Section 5.5.2.5.6.

5.3.3.5.4 shmdt()

A process invokes `shmdt()` to detach a shared memory region from its address space. DAC is performed by invoking the `ipcperms()` function.

5.3.3.5.5 shmctl()

A function that is invoked to set attributes, query status, or delete a shared memory region. A shared memory segment is not deleted until the last process detaches it. DAC is performed by invoking the `ipcperms()` function.

5.3.4 Signals

Signals offer a means of delivering asynchronous events to processes. Processes can send signals to each other with the `kill()` system call, or the kernel can internally deliver the signals. Events that cause a signal to be generated include keyboard interrupts via the interrupt, stop, or quit keys, exceptions from invalid instructions, or termination of a process. Signal transmission can be broken into two phases:

- Signal generation phase: The kernel updates appropriate data structures of the target process to indicate that a signal has been sent.
- Signal delivery phase: The kernel forces the target process to react to the signal by changing its execution state and or the execution of a designated signal handler is started.

Signal transmission does not create any user-visible data structures, so there are no object reuse issues. However, signal transmission does raise access control issues. This section describes relevant data structures and algorithms used to implement DAC.

5.3.4.1 Data structures

Access control is implemented in the signal generation phase. The main data structure involved in signal transmission access control is the process descriptor structure `task_struct`. The `task_struct` of each process contains fields that designate the real and effective user ID of the process for DAC access check. These fields are used to determine if one process is allowed to send a signal to another process.

5.3.4.2 Algorithms

Access control is performed at the signal generation phase. Signal generation, either from the kernel or from another process, is performed by invoking the routine `send_sig_info()`. The `kill()` system call, along with signal generation by the kernel, ultimately invokes `send_sig_info()`. `send_sig_info()` in turn calls `check_kill_permission()`, which allows signal generation if the kernel is trying to generate a signal for a process. For user processes, `send_sig_info()` delivers the signal after ensuring that at least one of the following is true:

- Sending and receiving processes belong to the same user.
- An administrator is the owner of the sending process.
- The signal is `SIGCONT` (to resume execution of a suspended process), and the receiving process is in the same login session of the sending process.

If one of the above three conditions are met, then DAC access is allowed. f the above conditions are not met, access is denied.

5.3.5 Sockets

A socket is an endpoint for communication. Two sockets must be connected to establish a communications link. Sockets provide a common interface to allow process communication across a network, such as an Internet domain, or on a single machine, such as a single UNIX domain.

Processes that communicate using sockets use a client-server model. A server provides a service, and clients make use of that service. A server that uses sockets first creates a socket and then binds a name to it. An Internet domain socket has an IP port address bound to it. The registered port numbers are listed in `/etc/services`. For example, the default port number for an ftp server is 21.

Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. The originator of the request, the client, creates a socket and makes a connection request on it, specifying the target address of the server. For an Internet domain socket, the address of the server is its IP address and its port number.

Sockets are created using the `socket()` system call. Depending on the type of socket, either UNIX domain or internet domain, the socket family operations vector invokes either `unix_create()` or `inet_create()`.

`unix_create()` and `inet_create()` invoke `sk_alloc()` to allocate the sock structure. `sk_alloc()` calls kmem_cache_alloc() to allocate memory, and then zeros the newly allocated memory by invoking `memset()`, thus taking care of object reuse issues associated with sockets created by users.

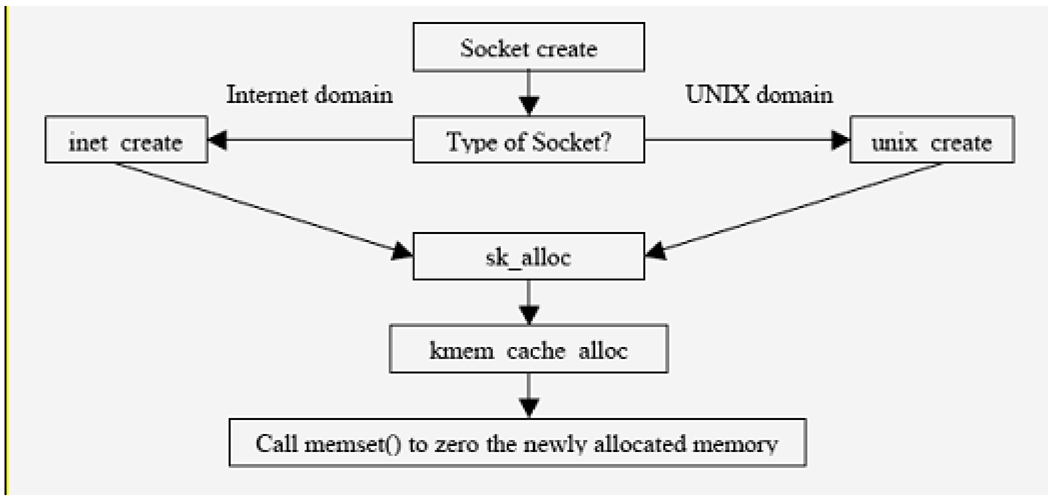


Figure 5-16: Object reuse handling in socket allocation

Calls to `bind()` and `connect()` to a UNIX domain socket file requires write access to it. UNIX domain sockets can be created in the ext3 file system, and therefore may have an ACL associated with them. For a more detailed description of client-server communication methods and the access control performed by them, refer to Section 5.12 of this document.

5.4 Network subsystem

The network subsystem allows Linux systems to connect to other systems over a network. It provides a general purpose framework within which network services are implemented. There are a number of possible hardware devices that can be connected, and a number of network protocols that can be used. The network subsystem abstracts both of these implementation details, so user processes and other kernel subsystems can access the network without knowing the physical devices and the protocol being used.

The various modules in the network subsystem are:

- Network device drivers communicate with the hardware devices. There is one device driver module for each possible hardware device.
- The device-independent interface module provides a consistent view of all of the hardware devices, so higher levels in the subsystem do not need specific knowledge of the hardware in use.
- The network protocol modules are responsible for implementing each of the possible network transport protocols.

- The protocol-independent interface module provides an interface that is independent of hardware devices and network protocol. This is the interface module that is used by other kernel subsystems to access the network without having a dependency on particular protocols or hardware. Finally, the system call interface module restricts the exported routines that user process can access.

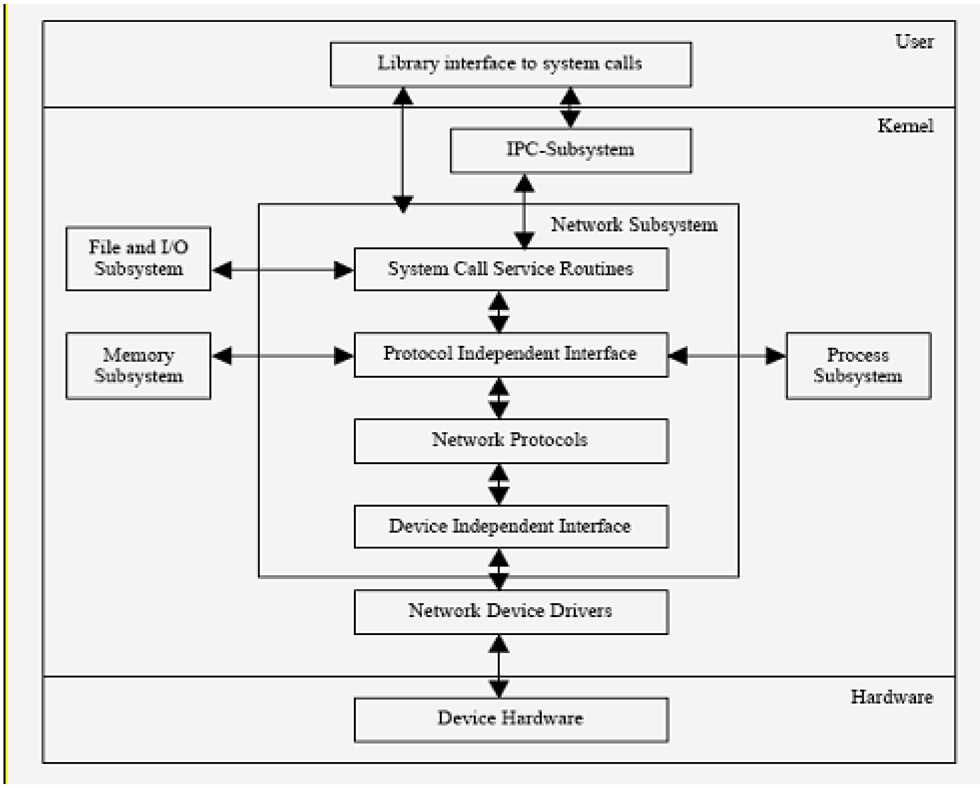


Figure 5-17: Network subsystem and its interaction with other subsystems

Network services include transmission and reception of data, network-independent support for message routing, and network-independent support for application software. The following subsections present an overview of the network stack and describe how various layers of the network stack are used to implement network services.

For more information, see the *TCP/IP Tutorial and Technical Overview* IBM Redbook by Adolfo, John & Roland. It is at the http://www.redbooks.ibm.com/abstracts/gg243376.html website.

5.4.1 Overview of the network protocol stack

The network protocol stack, which forms the carrier and pipeline of data from one host to another, is designed in such a way that one can interact with different layers at desired level. This section describes the movement of data through these stacked layers.

The physical layer and link layer work hand-in-hand. They consist of the network card and associated device driver, most often Ethernet but also Token Ring, PPP (for dial-up connections), and others.

The next layer, the network layer, implements the Internet Protocol, which is the basis of all Internet communications, along with related protocols, including Internet Control Message Protocol (ICMP).

The transport layer consists of the TCP, UDP and similar protocols.

The application layer consists of all the various application clients and servers, such as the Samba file and print server, the Apache web server, and others. Some of the application-level protocols include Telnet, for remote login; FTP, for file transfer; and, SMTP, for mail transfer.

Network devices form the bottom layer of the protocol stack. They use a link-layer protocol, usually Ethernet, to communicate with other devices to send and receive traffic. The interface put up by the network device driver copies packets from a physical medium, performs some error checks, and then puts up the packets to the network layer.

Output interfaces receive packets from the network layer, perform error checks, and then send the packets out over the physical medium. The main functionality of IP is routing:

- It checks incoming packets to see if they are for the host computer or if they need to be forwarded.
- It defragments packets if necessary and delivers them to the transport protocols.
- It has a dynamic database of routes for outgoing packets
- It addresses and fragments them if necessary before sending them down to the link layer.

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the most commonly used transport layer protocols. UDP simply provides a framework for addressing packets to ports within a computer, whereas TCP allows more complex connection-based operations such as the recovery of lost packets, and also traffic management implementations. Both UDP and TCP copy the application packet for transporting.

Moving up the transport layer, next is the INET (for internet) layer, which forms the intermediate layer between the transport layer and application sockets. The INET layer implements the sockets owned by the applications. All socket-specific operations are implemented here.

Each layer of the protocol stack adds a header containing layer-specific information to the data packet. A header for the network layer might include information such as source and destination addresses. The process of prepending data with headers is called encapsulation. Figure 5-18 shows how data is encapsulated by various headers. During decapsulation, the inverse occurs: the layers of the receiving stack extract layer-specific information and accordingly process the encapsulated data. Note that the process of encapsulation increases the overhead involved in transmitting data.

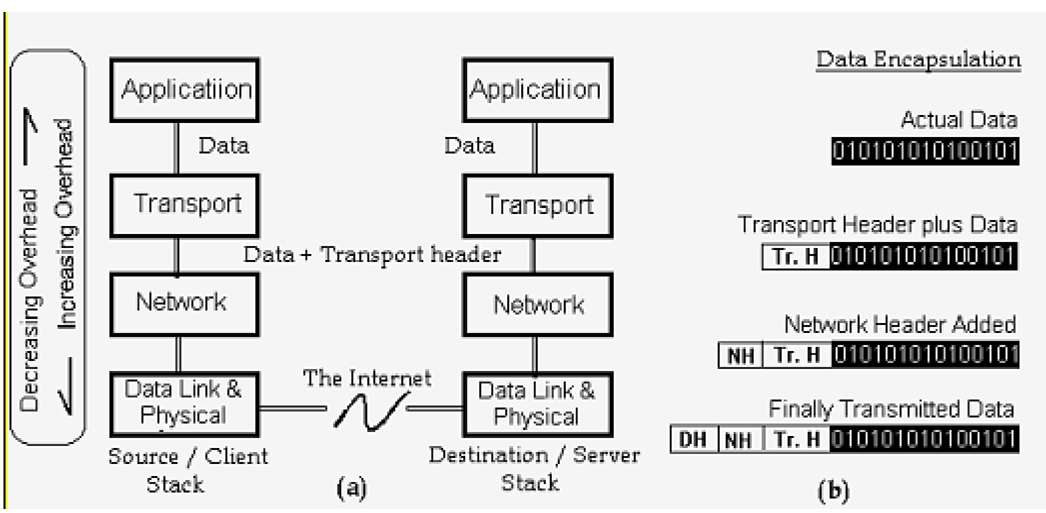


Figure 5-18: How data travels through the Network protocol stack

5.4.2 Transport layer protocols

The transport layer protocols supported by the SLES kernel are TCP and UDP.

5.4.2.1 TCP

TCP is a connection-oriented, end-to-end, reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. TCP provides for reliable IPC between pairs of processes in host computers attached to distinct but interconnected computer communication networks. TCP is used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data, or packets, that a message is divided into for efficient routing through the Internet. For more information about TCP, refer to RFC 793.

5.4.2.2 UDP

UDP is a connectionless protocol that, like TCP, runs on top of IP networks. UDP/IP provides very few error recovery services, offering instead a direct way to send and receive datagrams over an IP network. It is used primarily for broadcasting messages over a network. This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. For more information about UDP, look into RFC 768.

5.4.3 Network layer protocols

The network layer protocols supported by the SLES kernel are IP and ICMP.

5.4.3.1 Internet Protocol Version 4 (IPv4)

IPv4, aka simply IP, is the standard that defines the manner in which the network layers of two hosts interact. These hosts can be on the same network, or reside on physically distinct heterogeneous networks. In fact, IP was designed from the very beginning with inter-networking in mind.

IP provides a connectionless, unreliable, best-effort packet delivery service. Its service is called connectionless because in some ways it resembles the Postal Service. IP packets, like telegrams or mail messages, are treated independently. Each packet is stamped with the addresses of the receiver and the sender. Routing decisions are made on a packet-by-packet basis. IP is quite different from connection-oriented and circuit-switched phone systems that explicitly establish a connection between two users before any conversation, or data exchange, takes place, and maintain a connection for the entire length of exchange. For information about IP packets, IP addresses, and addressing formats refer to RFC 1883.

5.4.3.2 Internet Protocol Version 6 (IPv6)

The SLES kernel supports Internet Protocol version 6. IPv6 is the standard that defines the manner in which network layers of two hosts interact, and is an increment to existing IPv4. The TOE is in compliance with IPv6 source address selection as documented in RFC 3484, and implements several new socket options (`IPV6_RECVPKTINFO`, `IPV6_PKTINFO`, `IPV6_RECVHOPOPTS`, `IPV6_HOPOPTS`, `IPV6_RECVDSTOPTS`, `IPV6_DSTOPTS`, `IPV6_RTHDRDSTOPTS`, `IPV6_RECVRTHDR`, `IPV6_RTHDR`, `IPV6_RECVHOPOPTS`, `IPV6_HOPOPTS`, `IPV6_RECVTCLASS`) and ancillary data in order to support advanced IPv6 applications including `ping`, `traceroute`, routing daemons and others.

The following section introduces Internet Protocol Version 6 (IPv6). For additional information about referenced socket options and advanced IPv6 applications, see RFC 3542.

Internet Protocol Version 6 (IPv6) was designed to improve upon and succeed Internet Protocol Version 4 (IPv4).

IPv4 addresses consist of 32 bits. This accounts for about 4 billion available addresses. The growth of the Internet and the delegation of blocks of these addresses has consumed a large amount of the available address space. There has been a growing concern that someday we will run out of IPv4 addresses. IPv6 was an initiative to produce a protocol that improved upon the flaws and limitations of IPv4 and flexible enough to withstand future growth.

This introduction briefly addresses some of the features of IPv6. For further information, see RFC 2460, which contains the IPv6 specifications.

5.4.3.2.1 Addressing

IPv6 addresses are comprised of 128 bits, providing for more levels of an addressing hierarchy as well as space for future growth. A scope field has been added to multicast addresses to make for increased scalability. The scope identifies whether the packet should be multicast only on the link, site, or globally, which are levels of the addressing hierarchy.

IPv6 also introduces automatic configuration of IPv6 addresses. It uses the concept of network prefixes, interface identifiers, and MAC addresses to form and configure the host's IPv6 address. This IPv6 address is advertised to local IPv6 routers on the link, thus making for dynamic routing.

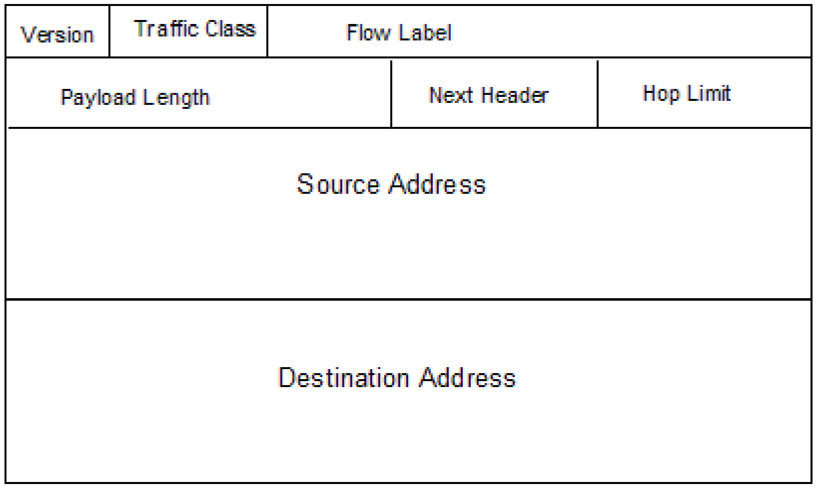
5.4.3.2.2 IPv6 Header

Some of the fields and options in an IPv4 header were removed from the IPv6 header. This helped to reduce the bandwidth required for IPv6, because the addresses themselves are larger. IP options are now placed in extension headers in IPv6. The extension header format is flexible and therefore will be easier to include additional options in the future. The extension headers are placed after the IP header and before any upper layer protocol fields.

RFC 2460 defines the following extension headers for IPv6:

- Hop-by-Hop
- Routing
- Fragment
- Destination Option
- Authentication
- Encapsulating Security Payload

IPv6 Header



5.4.3.2.3 Flow Labels

The IPv6 header has a field to in which to enter a flow label. This provides the ability to identify packets for a connection or a traffic stream for special processing.

5.4.3.2.4 Security

The IPv6 specifications mandate IP security. IP security must be included as part of an IPv6 implementation. IP security provides authentication, data integrity, and data confidentiality to the network through the use of the Authentication and Encapsulating Security Payload extension headers. IP security is described in more detail below.

5.4.3.3 Transition between IPv4 and IPv6

IPv4 addresses eventually need to be replaced with IPv6 ones. Due to the size of the Internet, it would be almost impossible to have a controlled roll out. So, IPv6 specifications incorporate a transition period in which IPv4 hosts and IPv6 hosts can communicate and reside together. RFC 4213 defines two mechanisms to accommodate the transition from IPv4 to IPv6, dual stack, and configured tunneling.

In a dual stack, both IPv4 and IPv6 are implemented in the operating system. Linux implements both IPv4 and IPv6.

The second mechanism uses tunnels. The IPv4 network exists while the IPv6 infrastructure is in progress. IPv6 packets are encapsulated in IPv4 packets and tunneled through IPv4 networks to final destination.

These mechanisms only accommodate the transition period that will be required as IPv6 infrastructure progresses and replaces IPv4. They allow for a more flexible deployment of IPv6.

5.4.3.4 IP Security (IPsec)

IP Security is an architecture developed for securing packets at the IP layer of the TCPIP protocol stack. It is comprised of several protocols and services, all working together to provide confidentiality, integrity, and authentication of IP datagrams.

The phrase data confidentiality refers to data that is secret or private, and that is read or seen only by the intended recipients. Unsecured packets traveling the internet can be easily intercepted by a network sniffer program and the contents viewed. The intended recipient would never know the received packet had been intercepted and read. The IP security architecture provides data confidentiality through encryption algorithms.

The phrase data integrity implies that the data received is as it was when sent. It has not been tampered, altered, or impaired in any way. Data authentication ensures that the sender of the data is really who you believe it to be. Without data authentication and integrity, someone can intercept a datagram and alter the contents to reflect something other than what was sent, as well as who sent it. IP Security provides data authentication and integrity through the use of hash functions in message authentication codes (HMACs).

The encryption algorithms and HMACs require several shared symmetric encryption keys. Thus IP Security also takes into consideration key management, and the secure exchange of keys through its services.

This introduction briefly describes the collection of protocols and services offered in IP Security.

5.4.3.4.1 Functional Description of IPsec

IP Security provides security at the IP Layer through the use of the AH and ESP protocols. These protocols operate in transport and tunnel mode.

In transport mode, AH and ESP provide security to the upper-layer protocols of the TCPIP protocol stack (that is, UDP and TCPIP). Therefore, only part of the IP datagram is protected. Transport mode is usually used for security between two hosts.

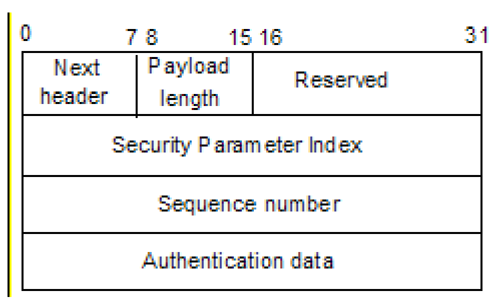
In tunnel mode, AH and ESP provide security to the entire IP datagram. The entire original IP datagram is encapsulated, and an outer IP header attached. Tunnel mode is usually for security between two gateways (that is, networks) or between a host and a gateway.

5.4.3.4.1.1 AH Protocol (AH)

The IP Authentication (AH) Header is described in RFC 2402. Besides providing data integrity and authentication of source, it also protects against replay attacks via the use of a sequence number and replay window.

The contents of the IP datagram, along with the shared secret key, are hashed, resulting in a digest. This digest is placed in the AH header, and the AH header is then included in the packet.

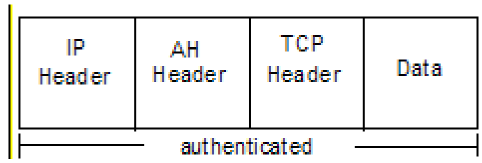
Verification occurs when the receiving end removes the AH header, then hashes its shared secret key with the IP datagram to produce a temporary digest and compare it with the digest in the AH header. If the two digests are identical, verification has succeeded.



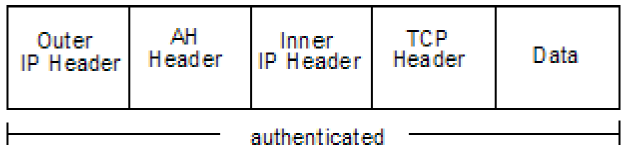
AH Header

When used in transport mode, the AH header is placed before the upper-layer protocol, which it will protect, and after the IP header and any options for IPv4. In the context of IPv6, according to RFC 2402, AH is viewed as an end-to-end payload. Therefore, the AH header should appear after the IP headers and hop-by-hop, routing, and fragmentation extension headers if present.

An IP Packet with transport mode AH



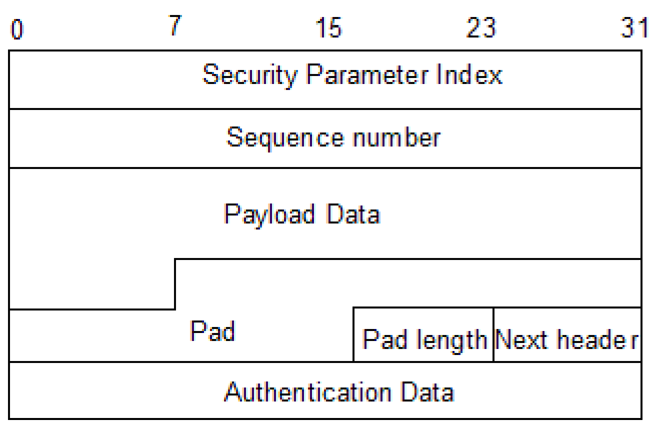
In tunnel mode, the entire IP datagram is encapsulated, protecting the entire IP datagram.



An IP Packet with tunnel mode AH

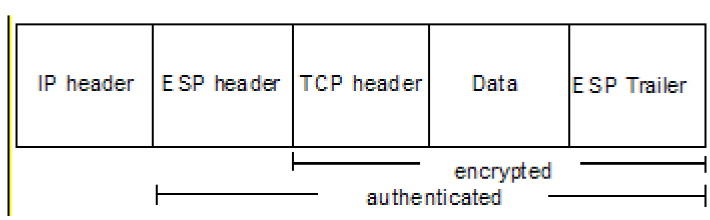
5.4.3.4.1.2 Encapsulating Security Payload Protocol (ESP)

The Encapsulating Security Payload (ESP) header is defined in RFC 2406. Besides data confidentiality, ESP also provides authentication and integrity as an option. The encrypted datagram is contained in the Data section of the ESP header. When authentication is also chosen within the ESP protocol, the data is encrypted first and then authenticated. The authenticated data is placed in the authentication data field. If no authentication is specified within the ESP protocol, then this field is not used.



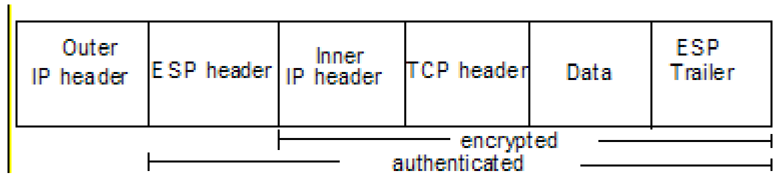
ESP Header

When used in transport mode, the ESP header is inserted after the IP header and before any upper-layer protocols, protecting only the upper layer protocols.



An IP Packet with transport mode ESP

In tunnel mode, the original IP header and any upper-layer protocols are encrypted and then authenticated if specified.



An IP Packet with tunnel mode ESP

5.4.3.4.1.3 Security Associations

RFC2401 defines a Security Association (SA) as a simplex or one-way connection that affords security services to the traffic it carries. Separate SAs must exist for each direction. IPSec stores the SAs in the Security Association Database (SAD), which resides in the Linux kernel.

5.4.3.4.1.4 Security Policy

A Security Policy is a general plan that guides the actions taken on an IP datagram. Upon processing an incoming or outgoing IP datagram, the security policy is consulted to determine what to do with the IP datagram. One of three actions will occur: the datagram is to be discarded, the datagram is to bypass IPSec processing, or IPSec processing is to be applied.

If IPSec processing is to be done, the policy will indicate which SA to access in the SAD. This SA will indicate the keys and service and other information related to the securing of the IP datagram. Security policies are stored in the Security Policy Database (SPD), which resides in the Linux kernel.

5.4.3.4.1.5 SA, SP, and Key Management

There are services for both manual and automated SA creation and maintenance. Manual management refers to an administrative user interface that allows for the creation and deletion of policy, SAs, and keys. Automated SA and key management is done through the IKE protocol.

5.4.3.4.1.6 Internet Key Exchange Protocol (IKE)

The Internet Key Exchange (IKE) protocol is designed so it can be used to negotiate SAs on behalf of any service, such as IPSec, RIPv2, or OSPF. Each service has a Domain of Interpretation (DOI) document describing the attributes required to establish an SA. IPSec's DOI is RFC 2407.

IKE has two basic phases, which it takes from Internet Security Association and Key Management Protocol (ISAKMP). The first phase establishes an authenticated and secure channel between two remote peers. During this phase the two peers exchange necessary information and keying material with each other. The result is an ISAKMP SA for each peer. This ISAKMP SA is then used to set up a secure and authenticated traffic stream when the two peers wish to negotiate an IPSec SA or an SA for any other service.

5.4.3.4.1.7 Socket API

The PF_KEY API, defined in RFC 2367, is a socket protocol family that allows userspace manual and automated key management tools to communicate with the kernel.

Linux also has `xfrm_user`, which makes use of the Netlink API, to communicate with the kernel.

The PF_KEY API defines interfaces for adding, deleting, updating, and retrieving SAs as well as maintaining the state and lifetime. The API has been extended in the Linux kernel to include SPD management too.

5.4.3.4.1.8 Cryptographic subsystem

IPSec uses the cryptographic subsystem described in this section. The cryptographic subsystem performs several cryptographic-related assignments, including Digital Signature Algorithm (DSA) signature verification, in-kernel key management, arbitrary-precision integer arithmetic, and verification of kernel modules signatures.

This subsystem was initially designed as a general-purpose mechanism, preserving the design ideas of simplicity and flexibility, including security-relevant network and file system services such as encrypted files and file systems, network file system security, strong file system integrity, and other kernel networking services where cryptography was required.

The ability to enforce cryptographic signatures on loadable modules has a couple of security uses:

- It prevents the kernel from loading corrupted modules
- It makes it difficult for an attacker to install a rootkit on a system

The kernel can be configured for checking or not checking the signatures of modules, so these signatures are only useful once the system is able to check it. For a signature to be checked and the new module accepted, it is first necessary that the kernel decrypt the signature with a public key. This public key is contained within the kernel, and the key must also to have the same checksum.

The in-kernel key management service allows cryptographic keys, authentication tokens, cross-domain user mappings, and other related security information to be cached in the kernel for the file systems to use other kernel services.

A special kind of key, called a keyring, which contains a list of keys and support links to others keys, is also permitted. The keys represent cryptographic data, authentication tokens, keyrings, and similar information.

The in-kernel key management service possesses two special types of keys: the above-mentioned keyring, and the user key. Userspace programs can directly create and manipulate keys and keyrings through a system call interface, using three new system calls: `add_key()`, `request_key()`, and `keyctl()`. Services can register types and search for keys through a kernel interface. There also exists an optional file system in which the key database can be manipulated and viewed.

For manipulating the key attributes and permissions it is necessary to be the key owner or to have administrative privileges.

5.4.4 Internet Control Message Protocol (ICMP)

Internet Control Message Protocol (ICMP) is an extension to IP that provides a messaging service. The purpose of these control messages is to provide feedback about problems in the communication environment. ICMP messages are sent in following situations:

- When a datagram cannot reach its destination.
- When the gateway does not have the buffering capacity to forward a datagram.
- When the gateway can direct the host to send traffic on a shorter route.

For more information about the ICMP, refer to RFC 792.

5.4.4.1 Link layer protocols

The Address Resolution Protocol (ARP) is the link layer protocol that is supported on the SLES system.

5.4.4.1.1 Address Resolution Protocol (ARP)

Address Resolution Protocol (ARP) is a protocol for mapping an IP address to a physical machine address that is recognized in the local network. For example, in IP Version 4, the most common level of IP in use today, an address is 32 bits long. In an Ethernet local area network, however, addresses for attached devices are 48 bits long. (The physical machine address is also known as a Media Access Control [MAC] address.) A table, usually called the ARP cache, is used to maintain a correlation between each MAC address and its corresponding IP address. ARP provides the protocol rules for making this correlation and providing address conversion in both directions.

There is also Reverse ARP (RARP), which can be used by a host to discover its IP address. In this case, the host broadcasts its physical address and a RARP server replies with the host's IP address.

5.4.5 Network services interface

The SLES kernel provides networking services through socket interfaces. When sockets are used to establish a connection between two programs across a network, there is always an asymmetry between the two ends. One end, on the server, creates a communication endpoint at a known address, and waits passively for connection requests. The other end, on the client, creates its own communication endpoint and actively connects to the server endpoint at its known address.

Figure 5-19 shows the steps taken by both server and client to establish a connection, along with the system calls used at each stage.

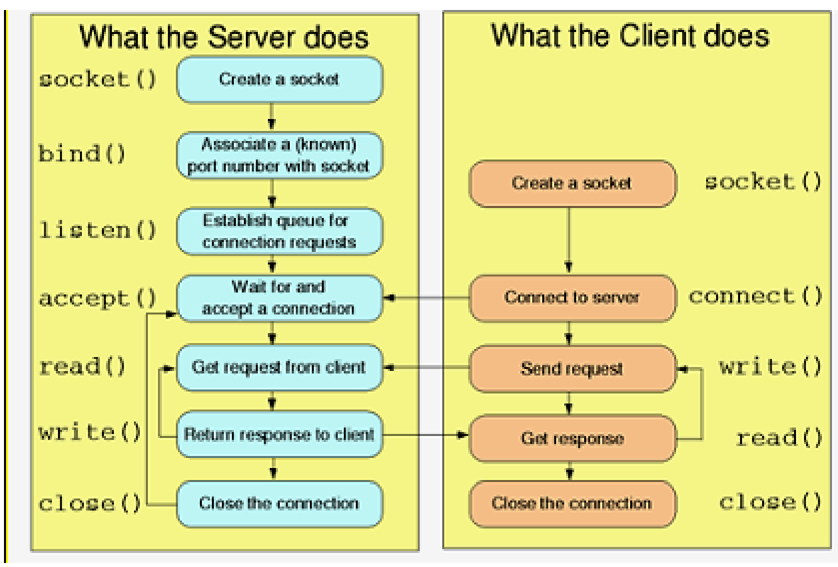


Figure 5-19: Server and client operations using socket interface

A communication channel is established using ports and sockets, which are needed to determine which local process at a given host actually communicates with which process, on which remote host, and using which protocol. A port is a 16-bit number, used by the host-to-host protocol to identify to which higher-level protocol or application program it must deliver incoming messages. Sockets, which are described in Section 5.3.5, are communications end points. Ports and sockets provide a way to uniquely and uniformly identify connections and the program and hosts that are engaged in them.

The following subsections describe access control and object reuse handling associated with establishing a communications channel.

5.4.5.1 socket()

`socket()` creates an endpoint of communication using the desired protocol type. Object reuse handling during socket creation is described in Section 5.3.5. `socket()` may perform additional access control checks by calling the `security_socket_create()` and `security_socket_post_create()` LSM hooks, but the SLES kernel does not use these LSM hooks.

5.4.5.2 bind()

`bind()` associates a name (address) to a socket that was created with the socket system call. It is necessary to assign an address to a socket before it can accept connections. Depending on the domain type of the socket, the bind function gets diverted to the domain-specific bind function.

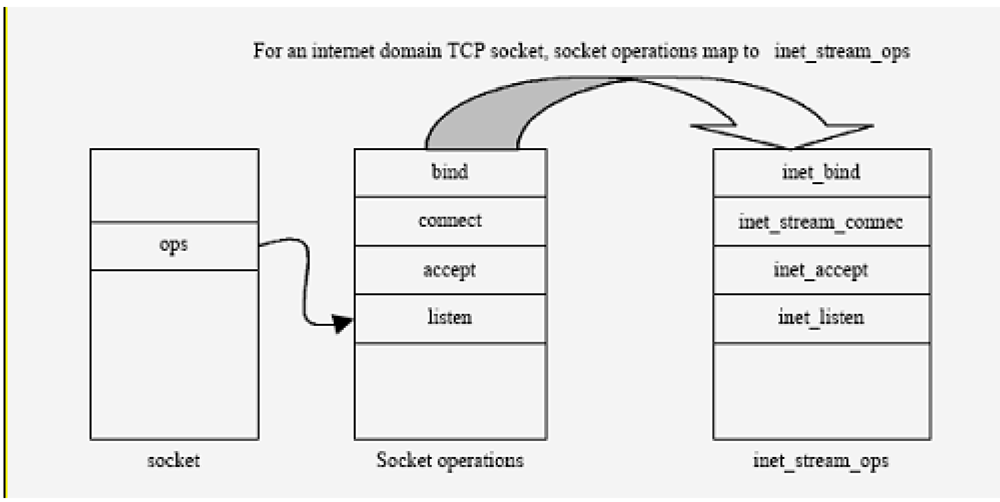


Figure 5-20: bind() function for internet domain TCP socket

If the port number being associated with a socket is below `PROT_SOCK` (defined at compile time as 1024), then `inet_bind()` ensures that the calling process possesses the `CAP_NET_BIND_SERVICE` capability. On the TOE, the `CAP_NET_BIND_SERVICE` capability maps to a uid of zero.

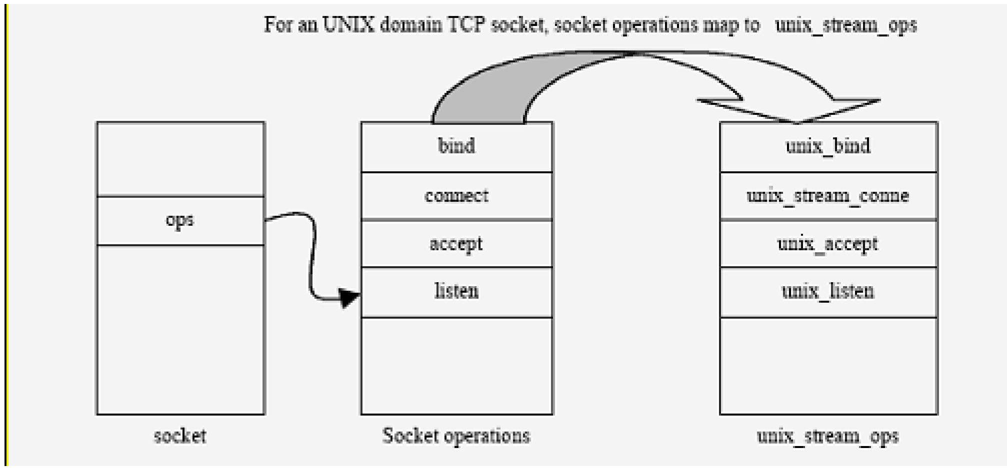


Figure 5-21: bind() function for UNIX domain TCP socket

Similarly, for UNIX domain sockets, `bind()` invokes `unix_bind()`. `unix_bind()` creates an entry in the regular ext3 file system space. This process of creating an entry for a socket in the regular file system space has to undergo all file system access control restrictions. The socket exists in the regular ext3 file system space, and honors DAC policies of the ext3 file system. `bind()` may perform additional access control checks by calling the `security_socket_bind()` LSM hook, but the SLES kernel does not use this LSM hook. `bind()` does not create any data objects that are accessible to users, so there are no object reuse issues to handle.

5.4.5.3 listen()

`listen()` indicates a willingness to accept incoming connections on a particular socket. A queue limit for the number of incoming connections is specified with `listen()`. Other than checking the queue limit, `listen()` does not perform DAC. It may perform additional access control checks by calling the `security_socket_listen()` LSM hook but the SLES kernel does not use this hook. `listen()` does not create any data objects that are accessible to users, so there are no object reuse issues to handle. Only TCP sockets support the `listen()` system call.

5.4.5.4 accept()

`accept()` accepts a connection on a socket. `accept()` does not perform any access control. `accept()` does not create any data objects that are accessible to users and therefore there are no object reuse issues to handle. Only TCP sockets support `accept()` system call.

5.4.5.5 connect()

`connect()` initiates a connection on a socket. The socket must be listening for connections; otherwise, the system call returns an error. Depending upon the type of the socket (stream for TCP or datagram for UDP), `connect()` invokes the appropriate domain type specific connection function. `connect()` does not perform DAC. It may perform additional access control checks by calling the `security_socket_connect()` LSM hook, but the SLES kernel does not use this hook. `connect()` does not create any data objects that are accessible to users, so there are no object reuse issues to handle.

5.4.5.6 Generic calls

`read()`, `write()` and `close()`: `read()`, `write()` and `close()` are generic I/O system calls that operate on a file descriptor. Depending on the type of object, whether regular file, directory, or socket, appropriate object-specific functions are invoked.

5.4.5.7 Access control

DAC mediation is performed at `bind()` time. The `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, `sendmsg()`, `recvmsg()`, `getsockname()`, `getpeername()`, `getsockopt()`, `setsockopt()`, and `shutdown()` syscalls may perform additional access control checks by calling LSM hooks but the SLES kernel does not do this. `read()`, `write()`, and `close()` operations on sockets do not perform any access control.

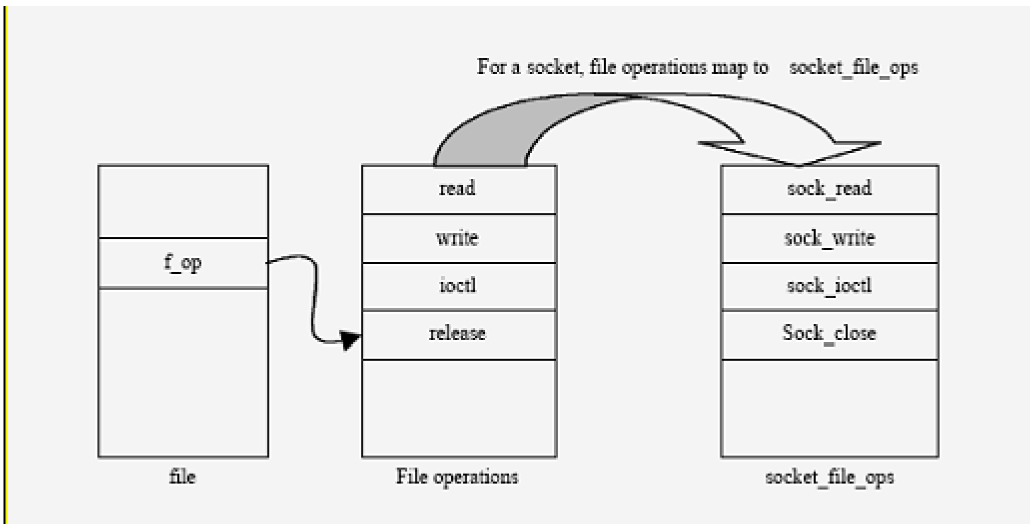


Figure 5-22: Mapping read, write and close calls for sockets

5.5 Memory management

The memory management subsystem is responsible for controlling process access to the hardware memory resources. This is accomplished through a hardware memory-management system that provides a mapping between process memory references and the machine's physical memory. The memory management subsystem maintains this mapping on a per-process basis, so two processes can access the same virtual memory address and actually use different physical memory locations. In addition, the memory management subsystem supports swapping; it moves unused memory pages to persistent storage to allow the computer to support more virtual memory than there is physical memory.

The memory management subsystem is composed of three modules:

- The architecture-specific module presents a virtual interface to the memory management hardware.
- The architecture-independent management module performs all of the per-process mapping and virtual memory swapping. This module is responsible for determining which memory pages will be evicted when there is a page fault; there is no separate policy module, since it is not expected that this policy will need to change.

- A system call interface is provided to provide restricted access to user processes. This interface allows user processes to allocate and free storage, and also to perform memory-mapped file I/O.

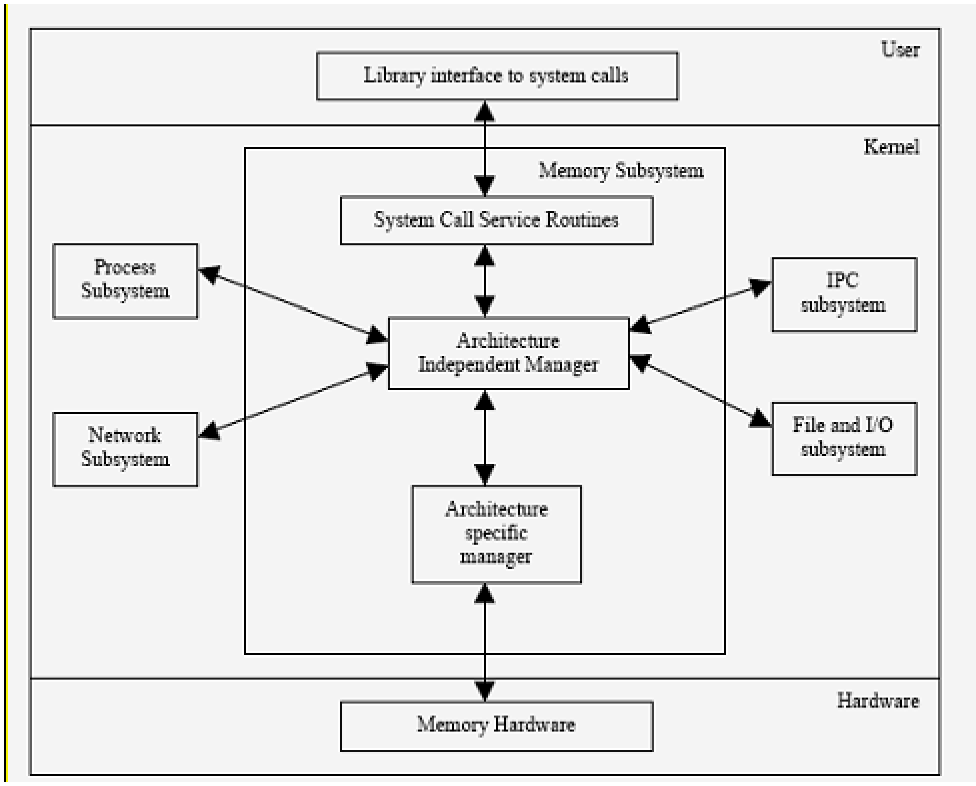


Figure 5-23: Memory subsystem and its interaction with other subsystems

This section highlights the implementation of the System Architecture requirements of a) allowing the kernel software to protect its own memory resources and b) isolating memory resources of one process from those of another, while allowing controlled sharing of memory resources between user processes.

This section is divided into five subsections. The first subsection, Four-Level Page Tables discuss recent changes to the Linux page table implementation. The second subsection, Memory Addressing, illustrates the SLES kernel's memory addressing scheme and highlights how segmentation and paging are used to prevent unauthorized access to a memory address. The third subsection, Kernel Memory Management, describes how the kernel allocates dynamic memory for its own use, and highlights how the kernel takes care of object reuse while allocating new page frames. The fourth subsection, Process Address Space, describes how a process views dynamic memory and what the different components of a process's address space are. The forth subsection also highlights how the kernel enforces access control with memory regions and handles object reuse with demand paging. The final subsection, Symmetric Multiprocessing and Synchronization, describes various SMP synchronization techniques used by the SLES kernel.

Because implementations of a portion of the memory management subsystem are dependent on the underlying hardware architecture, the following subsections identify and describe, where appropriate, how the hardware-dependent part of the memory management subsystem is implemented for the System x, System p, System z, and eServer 326 line of servers, which are all part of the TOE.

5.5.1 Four-Level Page Tables

Before the current implementation of four-level page tables, the kernel implemented a three-level page table structure for all architectures. The three-level page table structure that previously existed was constituted, from top to bottom, for the page global directory (PGD), page middle directory (PMD), and PTE.

In this implementation, the PMD is absent on systems that only present two-level page tables, so the kernel was able to recognize all architectures as if they possessed three-level page tables.

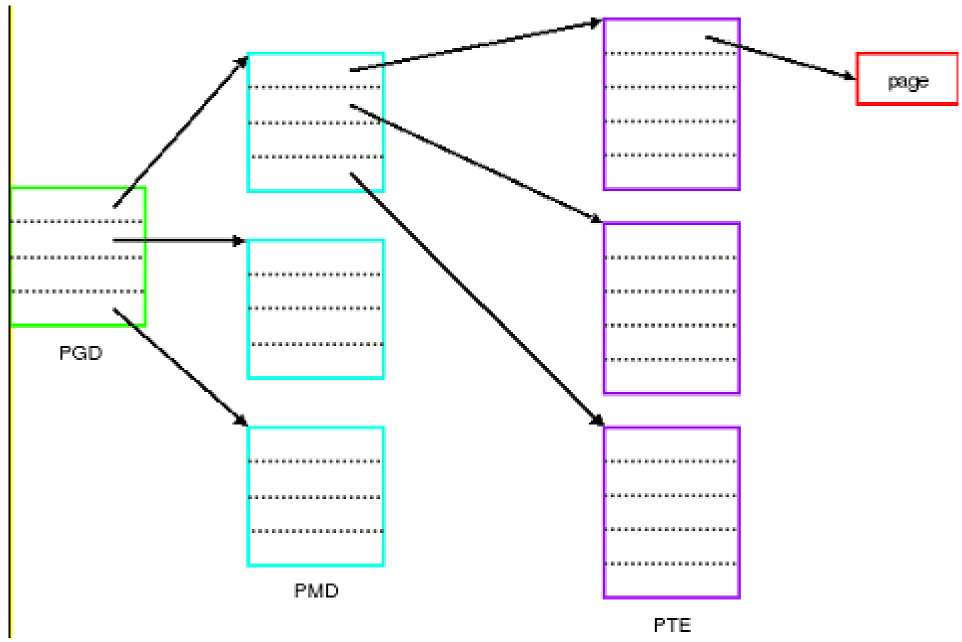


Figure 5-24: Previous three-level page-tables architecture

The new page table structure actually implemented includes a new level, called PUD, immediately below the top-level PGD directory. The PGD remains the top-level directory, and the PUD only exists on architectures that are using four-level page tables. The PMD and PTE levels present the same function as in previous kernels' implementations. Each of the levels existent in a page table hierarchy is indexed with a subset of the bits in the virtual address of interest.

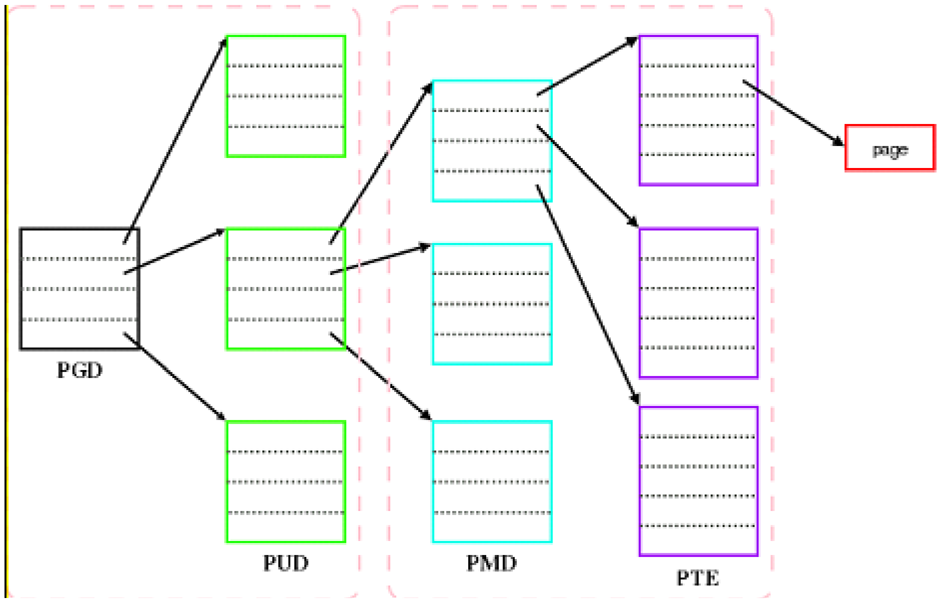


Figure 5-25: New page-table implementation: the four-level page-table architecture

The creation and insertion of a new level, the PUD level, immediately below the top-level PGD directory aims to maintain portability and transparency once all architectures have an active PGD at the top of hierarchy and an active PTE at the bottom. The PMD and PUD levels are only used in architectures that need them. These levels are optimized on systems that do not use them.

It is a less intrusive way to update the page tables' hierarchy, since it can be ignored by systems not using these levels. This is the same characteristic maintained by the kernel in the previous implementation of three-level page table architecture.

With this new implementation, architectures that use four-level page tables can have a virtual address space covering 128 TB of memory, far more than the 512 GB of virtual address space available with the old one.

5.5.2 Memory addressing

The main memory of a computer is a collection of cells that store data and machine instructions. Each cell is uniquely identified by a number or its memory address.

As part of executing a program, a processor accesses memory to fetch instructions or to fetch and store data. Addresses used by the program are virtual addresses. The memory management subsystem provides translation from virtual to real addresses. The translation process, in addition to computing valid memory locations, also performs access checks to ensure that a process is not attempting an unauthorized access.

Memory addressing is highly dependent on the processor architecture. The memory addressing for System x, System p, System z, and eServer 326 systems is described in the following sections.

5.5.2.1 System x

SLES provides enhanced handling of user process and kernel virtual address space for Intel x86-compatible systems (32 bit x86 systems only). Traditionally, 32-bit x86 systems had a fixed 4 GB virtual address space, which was allocated so the kernel had 1 GB and each user process 3 GB (referred to as the 3-1 split). This allocation has become restrictive because of the growing physical memory sizes. It is possible to configure a 4-4 split, where each user process and the kernel are allocated 4 GB of virtual address space. There are two important benefits to this new feature:

The larger kernel virtual address space allows the system to manage more physical memory. Up to 64 GB of main memory is supported by SLES on x86-compatible systems.

The larger user virtual address space allows applications to use approximately 30% more memory (3.7—3.8 GB), improving performance for applications that take advantage of the feature. This means that x86-compatible systems can be expected to have a longer life-span and better performance. FIXME unique?

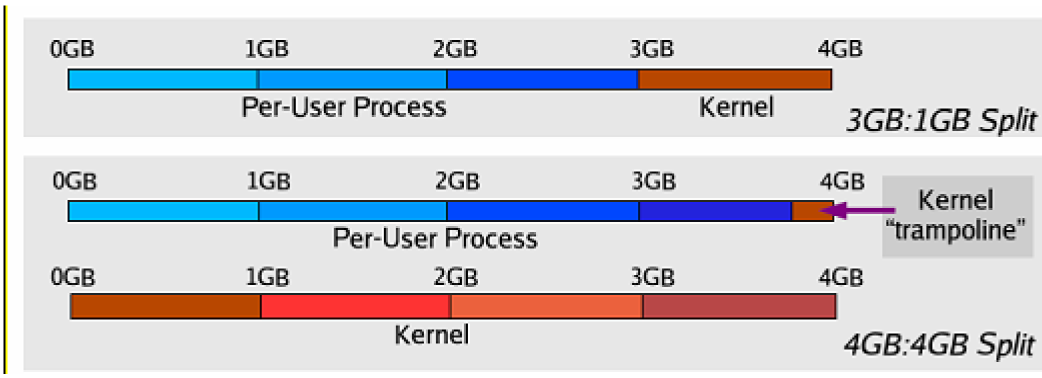


Figure 5-26: System x virtual addressing space

This section briefly explains the System x memory addressing scheme. The three kinds of addresses on System x are:

- Logical address: A logical address is included in the machine language instructions to specify the address of an operand or an instruction. It consists of a segment and an offset (or displacement) that denotes the distance from the start of the segment to the actual address.
- Linear address: A single 32-bit unsigned integer that can address up to 4 GB. That is, up to 4,294,967,296 memory cells.
- Physical address: A 32-bit unsigned integer that addresses memory cells in physical memory chips.

In order to access a particular memory location, the CPU uses its segmentation unit to transform a logical address into a linear address, and then a paging unit to transform a linear address into a physical address (see Figure 5-27).

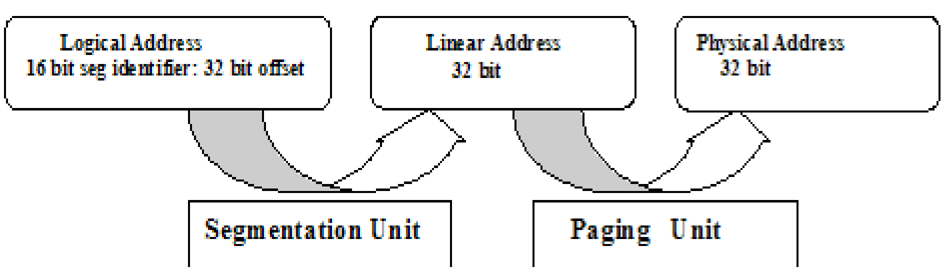


Figure 5-27: Logical Address Translation

5.5.2.1.1 Segmentation

The segmentation unit translates a logical address into a linear address. A logical address consists of two parts: a 16 bit segment identifier called the segment selector, and a 32-bit offset. For quick retrieval of the segment selector, the processor provides six segmentation registers whose purpose is to hold segment selectors. Three of these segmentation registers have specific purpose. For example, the code segment (cs) register points to a memory segment that contains program instructions. The cs register also includes a 2-bit field that specifies the Current Privilege Level (CPL) of the CPU. The CPL value of 0 denotes the highest privilege level, corresponding to the kernel mode; the CPL value of 3 denotes the lowest privilege level, corresponding to the user mode.

Each segment is represented by an 8-byte Segment Descriptor that describes the segment characteristics. Segment Descriptors are stored in either the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT). The system has one GDT, but may create an LDT for a process if it needs to create additional segments besides those stored in the GDT. The GDT is accessed through the GDTR processor register, while the LDT is accessed through the LDTR processor register.

From the perspective of hardware security access, both GDT and LDT are equivalent. Segment descriptors are accessed through their 16-bit segment selectors. A segment descriptor contains information, such as segment length, granularity for expressing segment size, and segment type, which indicates whether the segment holds code or data. Segment descriptors also contain a 2-bit Descriptor Privilege Level (DPL), which restricts access to the segment. The DPL represents the minimal CPU privilege level required for accessing the segment. Thus, a segment with a DPL of 0 is accessible only when the CPL is 0.

Figure 5-28 schematically describes access control as enforced by memory segmentation.

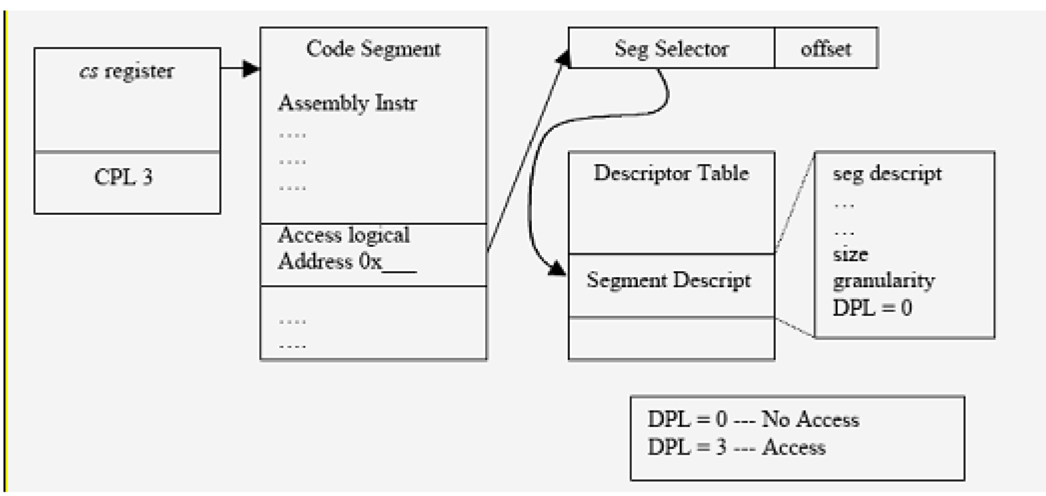


Figure 5-28: Access control through segmentation

5.5.2.1.2 Paging

The paging unit translates linear addresses into physical addresses. It checks the requested access type against the access rights of the linear address. Linear addresses are grouped in fixed-length intervals called pages. To allow the kernel to specify the physical address and access rights of a page instead of addresses and access rights of all the linear addresses in the page, continuous linear addresses within a page are mapped to continuous physical addresses.

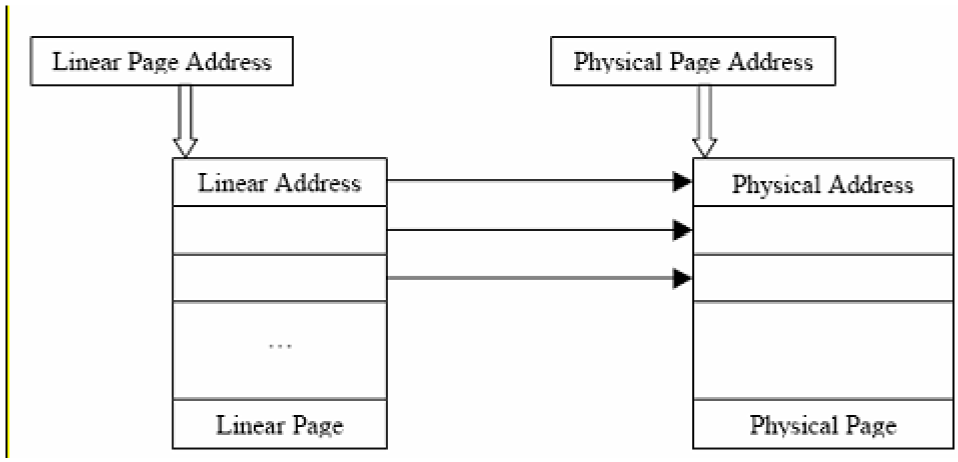


Figure 5-29: Contiguous linear addresses map to contiguous physical addresses

The paging unit sees all RAM as partitioned into fixed-length page frames. A page frame is a container for a page. A page is a block of data that can be stored in a page frame in memory or on disk. Data structures that map linear addresses to physical addresses are called page tables. Page tables are stored in memory and are initialized by the kernel when the system is started.

The System x supports two types of paging: regular paging and extended paging. The regular paging unit handles 4 KB pages, and the extended paging unit handles 4 MB pages. Extended paging is enabled by setting the Page Size flag of a Page Directory Entry.

In regular paging, 32 bits of linear address are divided into three fields:

- Directory: The most significant 10 bits represents directory.
- Table: The intermediate 10 bits represents table.
- Offset: The least significant 12 bits represents offset.

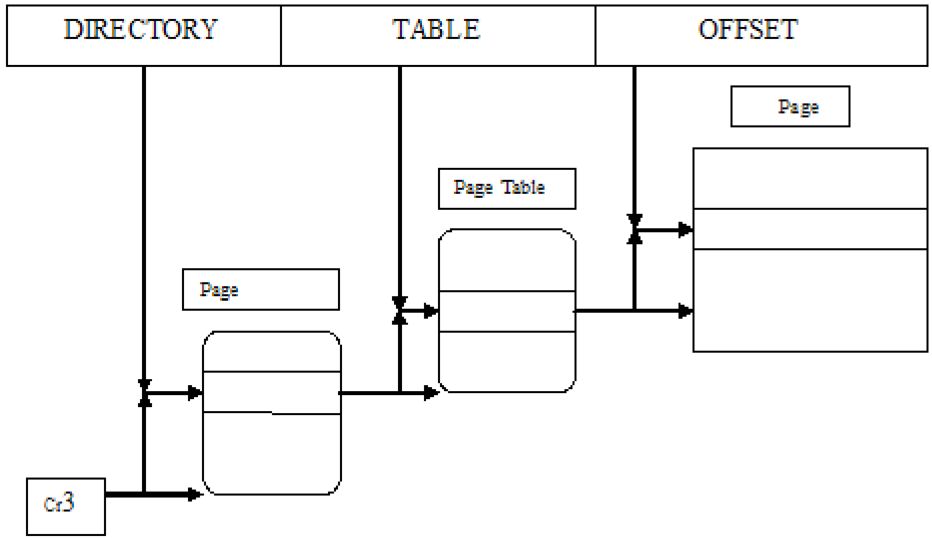


Figure 5-30: Regular paging

In extended paging, 32 bits of linear address are divided into two fields:

- Directory: The most significant 10 bits represents directory.
- Offset: The remaining 22 bits represents offset.

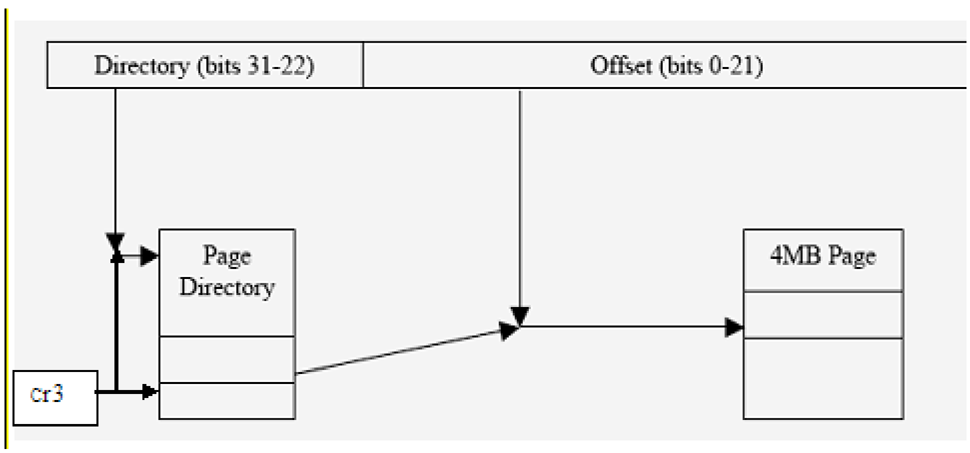


Figure 5-31: Extended paging

Each entry of the page directory and of the page table is represented by the same data structure. This data structure includes fields that describe the page table or page entry, such as accessed flag, dirty flag, and page size flag. The two important flags for access control are the Read/Write flag and the User/Supervisor flag.

Read/Write flag: Holds access rights of the page or the page table. The Read/Write flag is either read/write or read. If set to 0, the corresponding page or page table can only be read; otherwise, the corresponding page table can be written to or read.

User-Supervisor flag: This flag contains the privilege level that is required for accessing the page or page table. The User-Supervisor flag is either 0, which indicates that the page can be accessed only in kernel mode, or 1, which indicates that it can always be accessed.

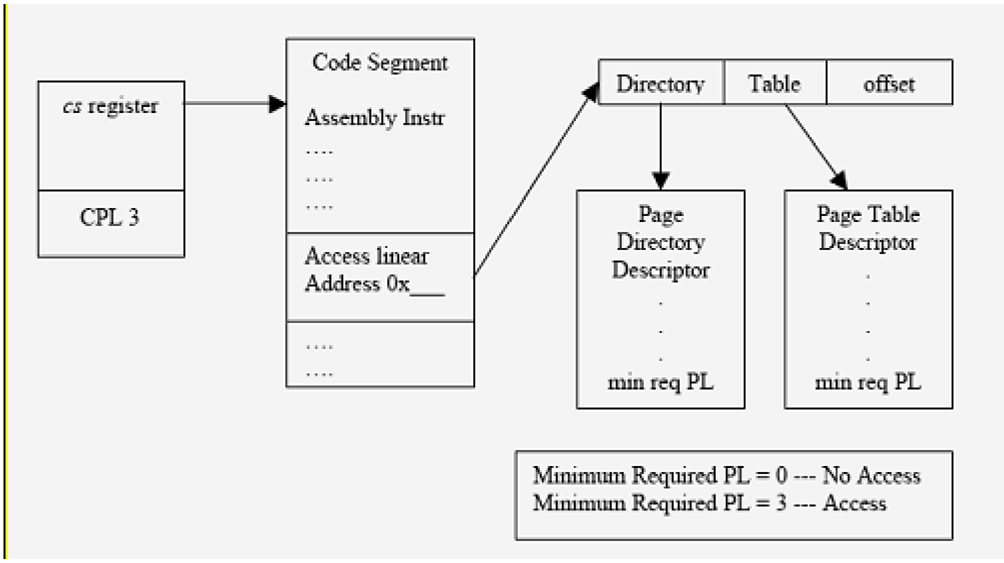


Figure 5-32: Access control through paging

5.5.2.1.2.1 Paging in the SLES kernel

The SLES kernel is based on Linux version 2.6.16, and implements three-level paging to support 64-bit architectures. The linear address is divided into the page global directory, the page middle directory, the page table, and the offset. On the TOE configuration of the SLES kernel running on System x systems, the page middle directory field is eliminated when it is set to zero.

5.5.2.1.2.2 Access control for control transfers through call gates

Call gates act as an interface layer between code segments at different privilege levels. They define entry points in more privileged code, to which control can be transferred. Intel processors use these call gates, which control CPU transitions from one level of privilege to other. Call gates are descriptors that contain pointers to code-segment descriptors and control access to those descriptors.

Operating systems can use call gates to establish secure entry points into system service routines. Before loading the code register with the code segment selector located in the call gate, the processor performs the following three privilege checks:

1. Compare the CPL with the call-gate DPL from the call-gate descriptor. The CPL must be less than or equal to the DPL.
2. Compare the RPL in the call-gate selector with the DPL. The RPL must be less than or equal to the DPL.
3. Call or jump, through a call gate, to a conforming segment requires that the CPL must be greater than or equal to the DPL. A call or jump through a call gate requires that the CPL must be equal to the DPL.

For more information about call gates, refer to the
http://www.csee.umbc.edu/~plusquel/310/slides/micro_arch4.html Web site.

5.5.2.1.2.3 Translation lookaside buffers

The System x processor includes other caches, in addition to the hardware caches. These caches are called Translation Lookaside Buffers (TLBs), and they speed up the linear-to-physical address translation. The TLB is built up as the kernel performs linear-to physical translations. Using the TLB, the kernel can quickly obtain a physical address corresponding to a linear address, without going through the page tables. Because address translations obtained from the TLB do not go through the paging access control mechanism, the kernel flushes the TLB buffer every time a process switch occurs between two regular processes. This process enforces the access control mechanism implemented by paging.

5.5.2.1.2.4 Address translations in 64-bit mode

All 16-bit and 32-bit address calculations are zero-extended in IA-32e mode to form 64-bit addresses. Address calculations are first truncated to the effective address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width. Because of this, 16-bit and 32-bit applications running in compatibility mode can only access the low 4 GBs of the 64-bit mode effective addresses. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4 GB of the 64-bit mode effective-address space.

5.5.2.1.2.5 Paging in EM64T

The 64-bit extensions architecture expands physical address extension (PAE) paging structures to potentially support mapping a 64-bit linear address to a 52-bit physical address. In the first implementation of the Intel EM64T, PAE paging structures are extended to support translation of a 48-bit linear address into a 40-bit physical address.

Prior to activating IA-32e mode, PAE must be enabled by setting CR4.PAE = 1. PAE expands the size of an individual page-directory entry (PDE) and page-table entry (PTE) from 32 bits to 64 bits to support physical-address sizes of greater than 32 bits. Attempting to activate IA-32e mode prior to enabling PAE results in a general-protection exception.

64-bit extensions architecture adds a new table, called the page map level 4 (PML4) table, to the linear-address translation hierarchy. The PML4 table sits above the page directory pointer (PDP) table in the page-translation hierarchy. The PML4 contains 512 eight-byte entries, with each entry pointing to a PDP table. Nine linear-address bits are used to index into the PML4.

PML4 tables are used in page translation only when IA-32e mode is activated. They are not used when IA-32e mode is disabled, regardless of whether or not PAE is enabled. The existing page-directory pointer table is expanded by the 64-bit extensions to 512 eight-byte entries from four entries. As a result, nine bits of the linear address are used to index into a PDP table rather than two bits. The size of both page-directory entry (PDE) tables and page-table entry (PTE) tables remains 512 eight-byte entries, each indexed by nine linear-address bits. The total of linear-address index bits into the collection of paging data structures (PML4 + PDP + PDE + PTE + page offset) defined above is 48. The method for translating the high-order 16 linear-address bits into a physical address is currently reserved.

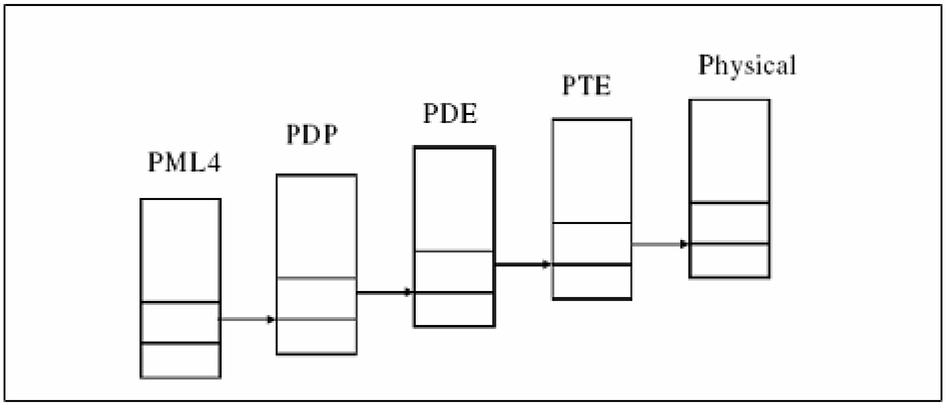


Figure 5-33: Paging data structures

The PS flag in the page directory entry (PDE.PS) selects between 4 KB and 2 MB page sizes.

5.5.2.2 System p

Linux on POWER5 System p systems runs only in Logical Partitioning (LPAR) mode. The System p offers the ability to partition one system into several independent systems through LPAR. LPAR divides the processors, memory, and storage into multiple sets of resources, so each set of resources can be operated independently with its own OS instance and applications.

LPAR allows multiple, independent operating system images of Linux to run on a single System p server. This section describes logical partitions and their impact on memory addressing and access control. To learn more about System p systems, see "PowerPC 64-bit Kernel Internals" by David Engebretson, Mike Corrigan & Peter Bergner at http://lwn.net/2001/features/OLS/pdf/pdf/ppc64.pdf.

System p systems can be partitioned into as many as 32 logical partitions. Partitions share processors and memory. A partition can be assigned processors in increments of 0.01. Figure represents a 4 CPU system that is split into 4 logical partitions. The hypervisor provides preemptive time slicing between partitions sharing a processor, guaranteeing that a partition gets the exact allocated share of the CPU, not more or less, even if the remainder of the processor is unused.

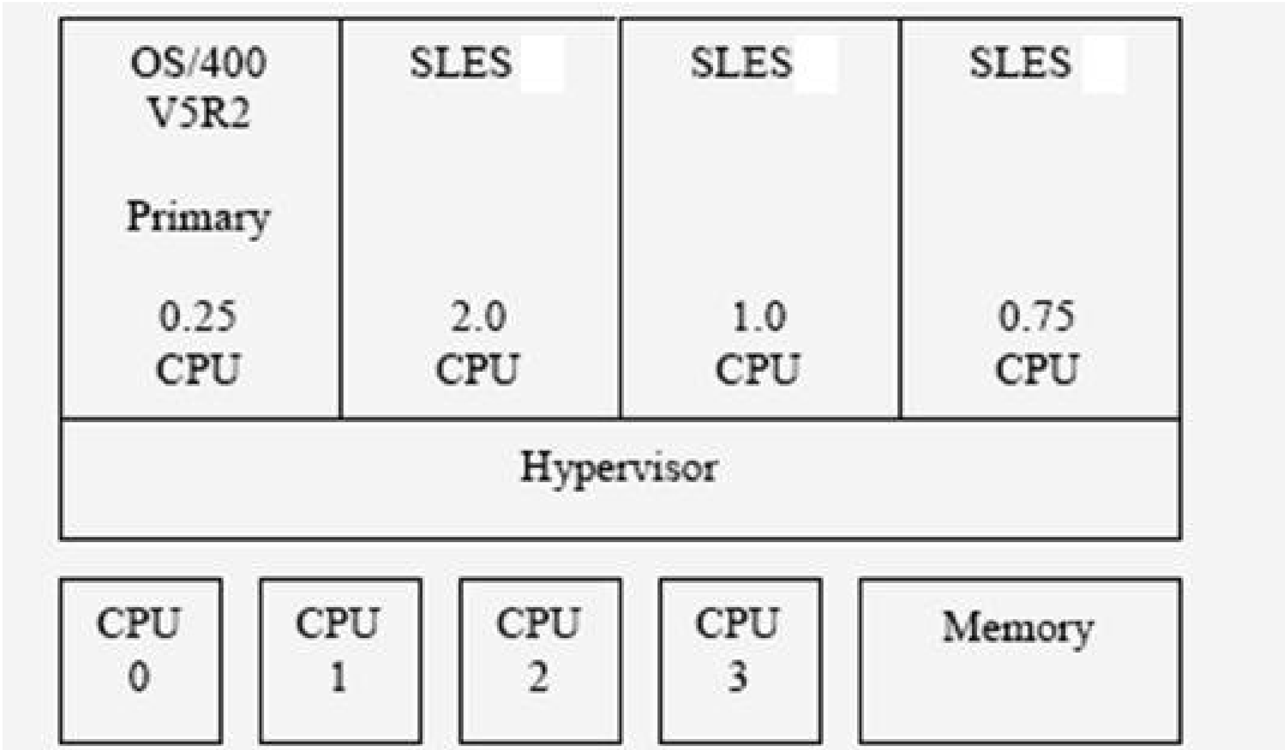


Figure 5-34: Logical partitions

On System p systems without logical partitions, the processor has two operating modes, user and supervisor. The user and supervisor modes are implemented using the PR bit of the Machine State Register (MSR). Logical partitions on System p systems necessitate a third mode of operation for the processor. This third mode, called the hypervisor mode, provides all the partition control and partition mediation in the system. It also affects access to certain instructions and memory areas. These operating modes for the processor are implemented using the PR and HV bits of the MSR.

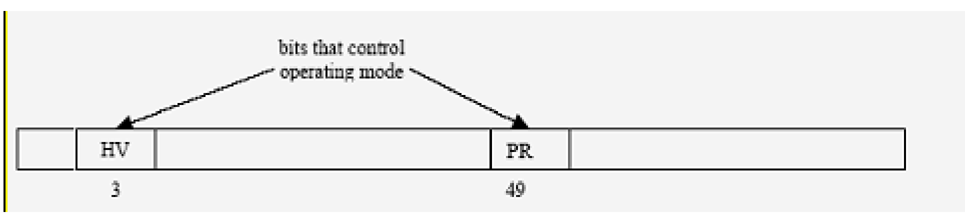


Figure 5-35: Machine state register

PR – Problem state.

- 0 The processor is in privileged state (supervisor mode).
- 1 The processor is in problem state (user mode).

HV – Hypervisor state.

- 0 The processor is not in hypervisor state.
- 1 If MSRPR= 0 the processor is in hypervisor state; otherwise, the processor is not in hypervisor state.

The hypervisor takes the value of 1 for hypervisor mode and 0 for user and supervisor mode. The following table describes the privilege state of the processor as determined by MSR [HV] and MSR [PR] as follows:

HV	**PR**	**Privilege State**
0	0	privileged(supervisor mode)
0	1	problem (user mode)
1	0	privileged and hypervisor
1	1	Problem(user mode)

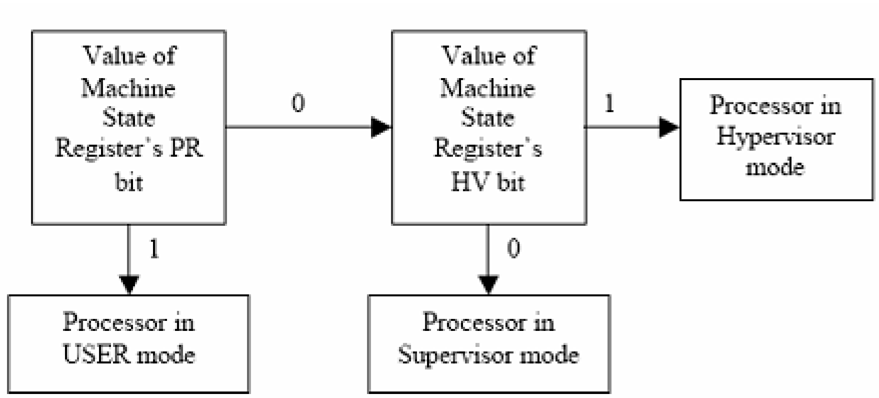


Figure 5-36: Determination of processor mode in LPAR

Just as certain memory areas are protected from access in user mode, some memory areas, such as hardware page tables, are accessible only in hypervisor mode. The PowerPC and POWER architecture provides only one system call instruction. This system call instruction, sc, is used to perform system calls from the user space intended for the SLES kernel, as well as hypervisor calls from the kernel space intended for the hypervisor. Hypervisor calls can only be made from the supervisor state. This access restriction to hypervisor calls is implemented with general purpose registers GPR0 and GPR3, as follows.

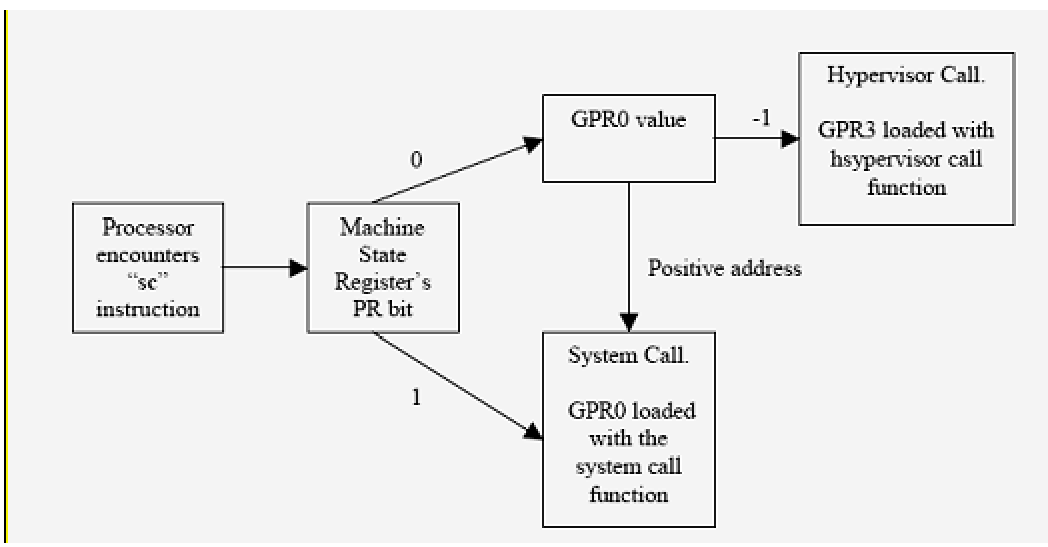


Figure 5-37: Transition to supervisor or hypervisor state

Actual physical memory is shared between logical partitions. Therefore, one more level of translation apart from the four levels described by System p section 5.5.2.2 is required to go from the effective address to the hardware address of the memory. This translation is done by the hypervisor, which keeps a logical partition unaware of the existence of other logical partitions.

5.5.2.2.1 Address Translation on LPARs

On System p systems running with logical partitions, the effective address, the virtual address, and the physical address format and meaning are identical to those of System p systems running in native mode. The kernel creates and translates them from one another using the same mechanisms described in Section 5.5.2.2. Access control by Block Address Translation and Page Address Translation, described in Section 5.5.2.2, and are performed here as well.

The Block Address Translation and Page Address Translation mechanisms provide System p logical partitions with the same block and page level memory protection capabilities, granular to no-access, read access, and read-write access. These capabilities allow the majority of the kernel code to remain common between System p native mode and System p LPAR mode.

5.5.2.2.2 Hypervisor

The hypervisor program is stored in a system flash module in the server hardware. During system initialization, the hypervisor is loaded into the first physical address region of system memory. The hypervisor program is trusted to create partition environments, and is the only program that can directly access special processor registers and translation table entries. Partition programs have no way to access the hypervisor instructions or data, other than through controlled hypervisor service calls that are part of the processor architecture. These protections allow the hypervisor to perform its duties in a simple and rigorous manner, resulting in the confinement of each operating system to a very tight, inescapable box.

Because the hypervisor is accessible only through the kernel mode, no specific access control is performed when the kernel interacts with the hypervisor. The kernel does provide an RTAS system call to authorized programs for interacting with the hardware. Run time abstraction services (RTAS) is a firmware interface that shields the operating system from details of the hardware. The RTAS ensures that the calling process possesses the `CAP_SYS_ADMIN` capability.

5.5.2.2.3 Real mode addressing

Each operating system image requires a range of memory that can be accessed in real addressing mode. In this mode, no virtual address translation is performed, and addresses start at address 0. Operating systems typically use this address range for startup kernel code, fixed kernel structures, and interrupt vectors. Since multiple partitions cannot be allowed to share the same memory range at physical address 0, each partition must have its own real mode addressing range.

As each partition is started, the hypervisor assigns that partition a unique real mode address offset and range value, and then sets these offset and range values into registers in each processor in the partition. These values map to a physical memory address range that has been exclusively assigned to that partition.

When partition programs access instructions and data in real addressing mode, the hardware automatically adds the real mode offset value to each address before accessing physical memory. In this way, each logical partition programming model appears to have access to physical address 0, even though addresses are being transparently redirected to another address range. Hardware logic prevents modification of these registers by operating system code running in the partitions. Any attempt to access a real address outside the assigned range results in an addressing exception interrupt, which is handled by the operating system exception handler in the partition.

5.5.2.2.4 Virtual mode addressing

Operating systems use another type of addressing, virtual addressing, to give user applications an effective address space that exceeds the amount of physical memory installed in the system. The operating system does this by paging infrequently used programs and data from memory out to disk, and bringing them back into memory on demand.

When applications access instructions and data in virtual addressing mode, they are not aware that their addresses are being translated by virtual memory management using page translation tables. These tables reside in system memory, and each partition has its own exclusive page table. Processors use these tables to transparently convert the virtual address for a program into the physical address where that page has been mapped into physical memory. In this case, if the page has been moved out of physical memory onto disk, the operating system receives a page fault. In a logical partitioning operation, the page translation tables are placed in reserved physical memory regions that are only accessible to the hypervisor. In other words, the page table for a partition is located outside the real mode address range of that partition. The register that provides the physical address of its page table to a processor is also protected by hardware logic, so it cannot be modified by partition programs, and can only be modified by the hypervisor.

When the operating system needs to create a page translation mapping, it must execute a specially designed hypervisor service call instruction on one of its processors, which transfers execution to a hypervisor program. The hypervisor program creates the page table entry on behalf of the partition, and adds a logical-to-physical offset to the table entry before storing it. Partition programs can also make hypervisor calls to modify or delete existing page table entries. Page table entries only map into specific physical memory regions, called logical address regions, which have been assigned in granular chunks to that partition. These logical address regions provide the physical memory that backs up the virtual page address spaces for the partition. Memory for a partition, therefore, is composed of one contiguous real mode addressing region, plus some number of logical address regions, which can be assigned in any order from anywhere in memory.

5.5.2.2.5 Access to I/O address space

In addition to mapping virtual page addresses into the physical memory for a partition, the operating system can make hypervisor calls to map page table addresses into the physical register and buffer address spaces on PCI I/O adapters. Device drivers directly read and write these adapter registers and buffers, which is how they set up and control I/O operations on the PCI devices. In LPAR, the PCI I/O adapter must be assigned to a given partition before that operating system can make a mapping request service call; otherwise, the hypervisor will not permit the creation of the page table entry.

5.5.2.2.6 Direct Memory Access addressing

PCI I/O adapter direct memory access (DMA) operations move data between I/O adapters and system memory, and they use a similar address relocation mechanism to page tables. PCI host bridge hardware translates addresses generated by I/O devices into physical memory addresses.

The I/O bridge makes this translation with a translation control entry (TCE) table, which is also stored in physical memory. As with page tables, this TCE table resides in a physical address region of system memory that is inaccessible by partitions, and only accessible by the hypervisor. Unlike page tables, however, TCE tables are not tied to any one partition. By calling a hypervisor service, partition programs can create, modify, or delete TCE table entries for the specific PCI adapters assigned to that partition. The hypervisor adds a physical address offset to the DMA target address provided by the partition program. When the I/O bridge translates an I/O adapter DMA address into physical memory, the resulting address falls within the physical memory space assigned to that partition.

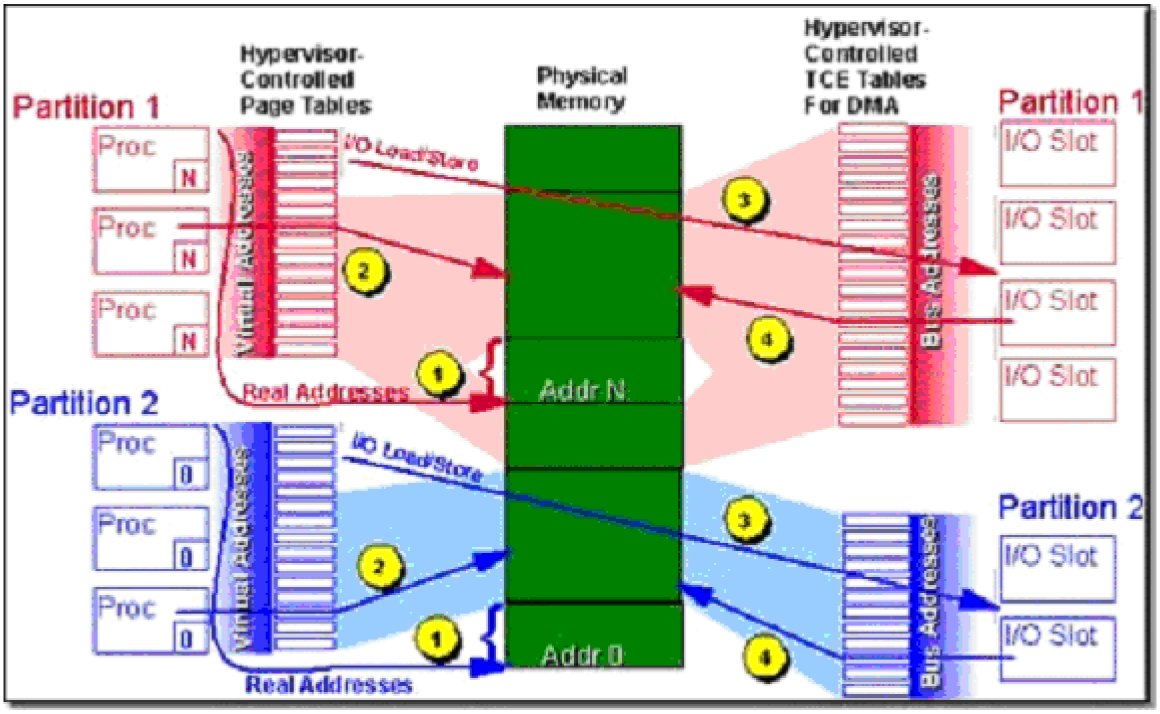


Figure 5-38: DMA addressing

5.5.2.2.7 Run-Time Abstraction Services

System p hardware platforms provide a set of firmware Run-Time Abstraction Services (RTAS) calls. In LPAR, these calls perform additional validation checking and resource virtualization for the partitioned environment. For example, although there is only one physical non-volatile RAM chip, and one physical battery-powered Time-of-Day chip, RTAS makes it appear to each partition as though it has its own non-volatile RAM area, and its own uniquely settable Time-of-Day clock. Because RTAS calls run inside a partition with the operating system, even they are not allowed to access anything outside the partition without a call to the hypervisor.

5.5.2.2.8 Preventing denial of service

LPARs run on top of an advanced symmetrical multiprocessor architecture, and some resources are also implicitly shared by the partitions. The hypervisor is designed to keep partitions from using shared resources in a way that would deny or restrict access to those resources by other partitions. A key example is the hypervisor itself, which is implemented as a library of services shared by the partitions. These services are called within the context of the operating system running in each partition. Each service call is dispatched on the specific processor from which the call was made, so hypervisor calls execute only on processors owned by the calling partition. Therefore, regardless of the type and frequency of hypervisor calls made by a partitioned operating system, they can have no effect on hypervisor usage or access in other partitions.

In a similar fashion, the hypervisor and hardware mechanisms protect shared hardware resources. No operation within a partition can take exclusive control of a shared hardware resource, or use a shared resource in a way that inhibits other partitions' access.

5.5.2.3 System p native mode

This section describes memory addressing and memory management of System p in native mode. Note that SLES runs on System p in LPAR mode.

For further information about PowerPC 64 bit processor, see PowerPC 64-bit Kernel Internals by David Engebretson, Mike Corrigan & Peter Bergner at http://lwn.net/2001/features/OLS/pdf/pdf/ppc64.pdf. You can find further in formation about System p hardware at http://www-1.ibm.com/servers/eserver/pseries/linux/.

The following describes the four address types used in System p systems. They are effective, virtual, physical, and block:

- Effective address: The effective address, also called the logical address, is a 64-bit address included in the machine language instruction of a program to fetch an instruction, or to fetch and store data. It consists of an effective segment ID (bits 0-35), a page offset within the segment (bits 36-51), and a byte offset within the page (bits 52-63).

Effective segment ID (ESID)	Page Offset	Byte Offset

Figure 5-39: Effective address

- Virtual address: The virtual address, which is equivalent to the linear address of System x, is a 64-bit address used as an intermediate address while converting an effective address to a physical address. It consists of a virtual segment ID (bits 0-35), a page offset within the segment (bits 36-51), and a byte offset within the page (bits 52-63). All processes are given a unique set of virtual addresses. This allows a single hardware page table to be used for all processes. Unique virtual addresses for processes are computed by concatenating the effective segment ID (ESID) of the effective address with a 23-bit field, which is the context number of a process. All processes are defined to have a unique context number. The result is multiplied by a large constant and masked to produce a 36-bit virtual segment ID (VSID). In case of kernel addresses, the high order nibble is used in place of the context number of the process.

Virtual segment ID (VSID)	Page Offset	Byte Offset

Figure 5-40: Virtual address

- Physical address: The physical address is a 64-bit address of a memory cell in a physical memory chip.
- Block address: A block is a collection of contiguous effective addresses that map to contiguous physical addresses. Block sizes vary from 128 KB to 256 MB. The block address is the effective address of a block.

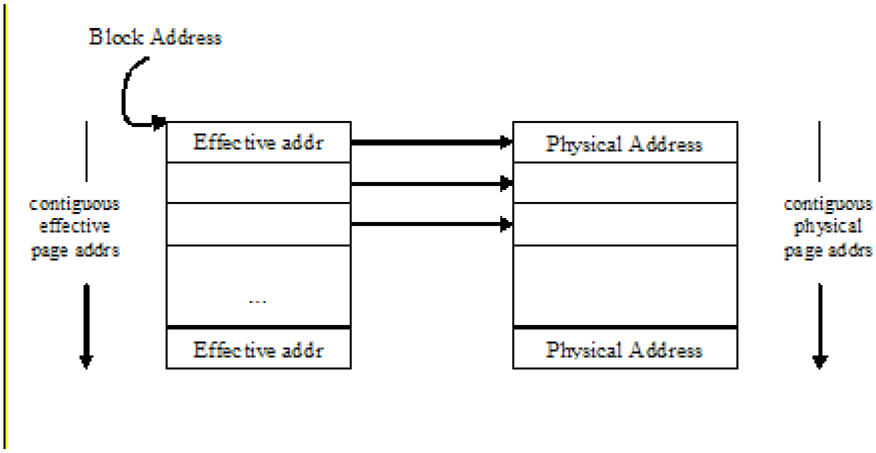


Figure 5-41: Block address

- To access a particular memory location, the CPU transforms an effective address into a physical address using one of the following address translation mechanisms.
- Real mode address translation, where address translation is disabled. The physical address is the same as the effective address.
- Block address translation, which translates the effective address corresponding to a block of 128 KB to 256 MB size.
- Page address translation, which translates a page-frame effective address corresponding to a 4-Kbyte page.

The translation mechanism is chosen based on the type of effective address (page or block) and settings in the processor Machine State Register (MSR).

Settings in the MSR and page, segment, and block descriptors are used in implementing access control. The following describes the MSR, page descriptor, segment descriptor, and block descriptor structures, and identifies fields that are relevant for implementing access control.

5.5.2.3.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. The MSR defines the state of the processor.

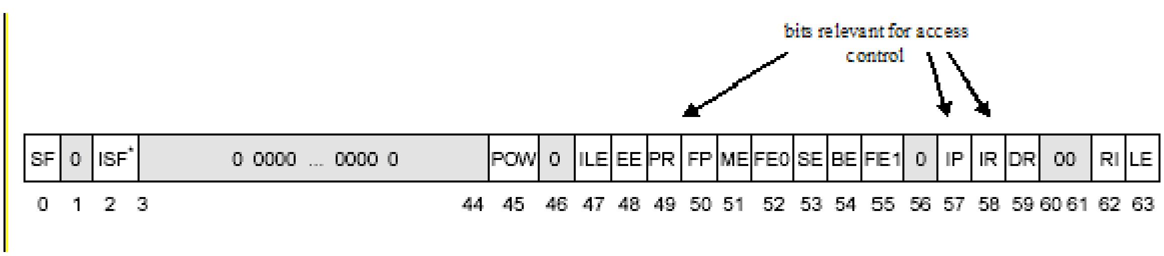


Figure 5-42: Machine state register

- PR: Privilege Level. The Privilege Level takes the value of 0 for the supervisor level and 1 for the user level.
- IR: Instruction Address Translation. The value of 0 disables translation, and the value of 1 enables translation.

- DR: Data Address Translation. The value of 0 disables translation, and the value of 1 enables translation.

5.5.2.3.2 Page descriptor

Pages are described by Page Table Entries (PTEs). The operating system generates and places PTEs in a page table in memory. A PTE on SLES is 128 bits in length. Bits relevant to access control are Page protection bits (PP), which are used with MSR and segment descriptor fields to implement access control.

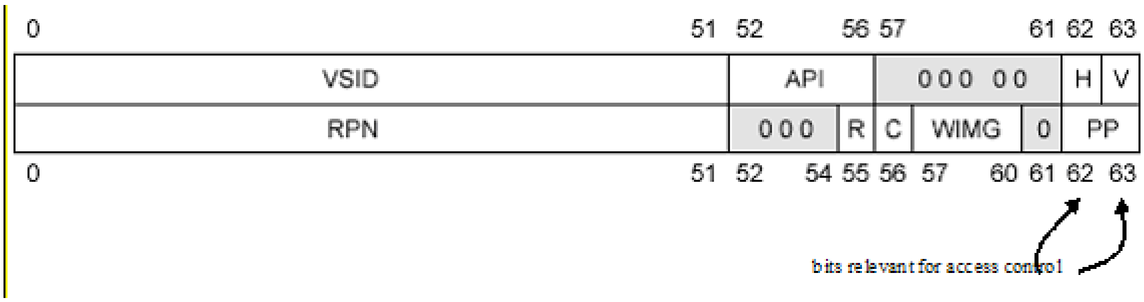


Figure 5-43: Page table entry

5.5.2.3.3 Segment descriptor

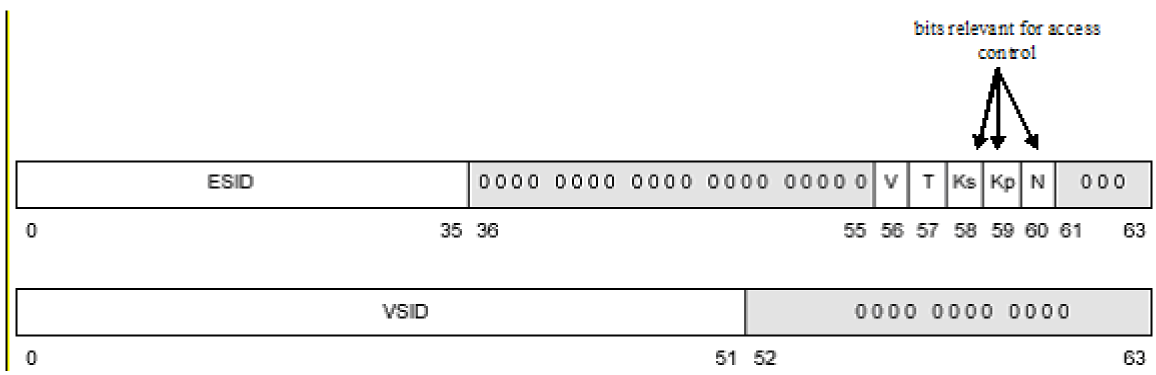


Figure 5-44: Segment Table Entry

Segments are described by Segment Table Entries (STEs). The operating system generates and places STEs in segment tables in memory. Each STE is a 128-bit entry that contains information for controlling segment search process, and for implementing the memory-protection mechanism.

Ks: Supervisor-state protection key

- Kp: User-state protection key
- N: No-execute protection bit

5.5.2.3.4 Block descriptor

For address translation, a pair of special-purpose registers called upper and lower BAT (Block Address Translation) registers, which contain effective and physical addresses for the block, define each block.

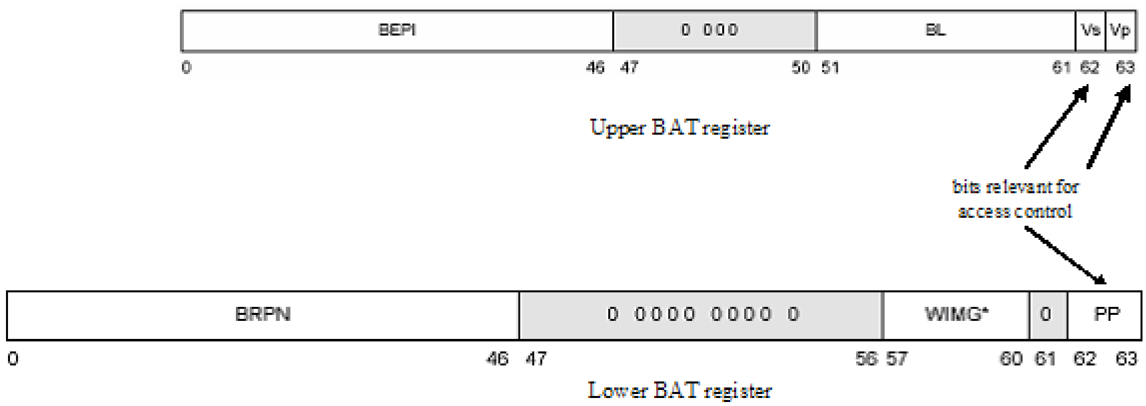


Figure 5-45: Block Address Translation entry

- Vs: Supervisor mode valid bit. Used with MSR[PR] to restrict translation for some block addresses.
- Vp: User mode valid bit. Used with MSR[PR] to restrict translation for some block addresses.
- PP: Protection bits for block.

5.5.2.3.5 Address translation mechanisms

The following simplified flowchart describes the process of selecting an address translation mechanism based on the MSR settings for instruction (IR) or data (DR) access. For performance measurement, the processor concurrently starts both Block Address Translation (BAT) and Segment Address Translation. BAT takes precedence; therefore, if BAT is successful, Segment Address Translation result is not used.

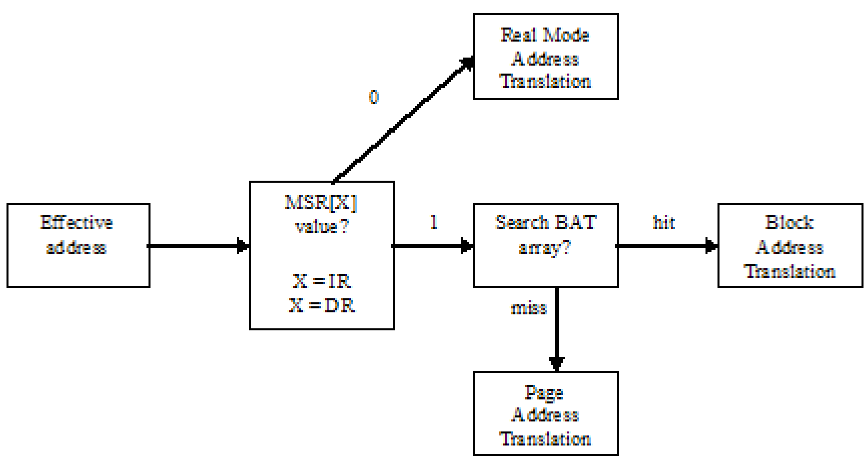


Figure 5-46: Address translation method selection

The following sections describe the three address translation mechanisms, and the access controls they perform.

Real Mode Address Translation: Real Mode Address Translation is not technically the translation of any addresses. Real Mode Address Translation signifies no translation. That is, the physical address is the same as the effective address. The operating system uses this mode during initialization and some interrupt processing. Because there is no translation, there is no access control implemented for this mode. However, because only the super user can alter MSR[IR] and MSR[DR], there is no violation of security policy.

Block Address Translation (BAT) and access control: BAT checks to see if the effective address is within a block defined by the BAT array. If it is, BAT goes through the steps described in to perform the access check for the block and get its physical address.

BAT allows an operating system to designate blocks of memory for use in user mode access only, for supervisor mode access only, or for user and supervisor access. In addition, BAT allows the operating system to protect blocks of memory for read access only, read-write access, or no access.

BAT treats instruction or data fetches equally. That is, using BAT, it is not possible to protect a block of memory with the no-execution access (no instruction fetches, only data load and store operations allowed). Memory can be protected with the no-execution bit on a per-segment basis, allowing the PAT mechanism to implement access control based on instruction or data fetches.

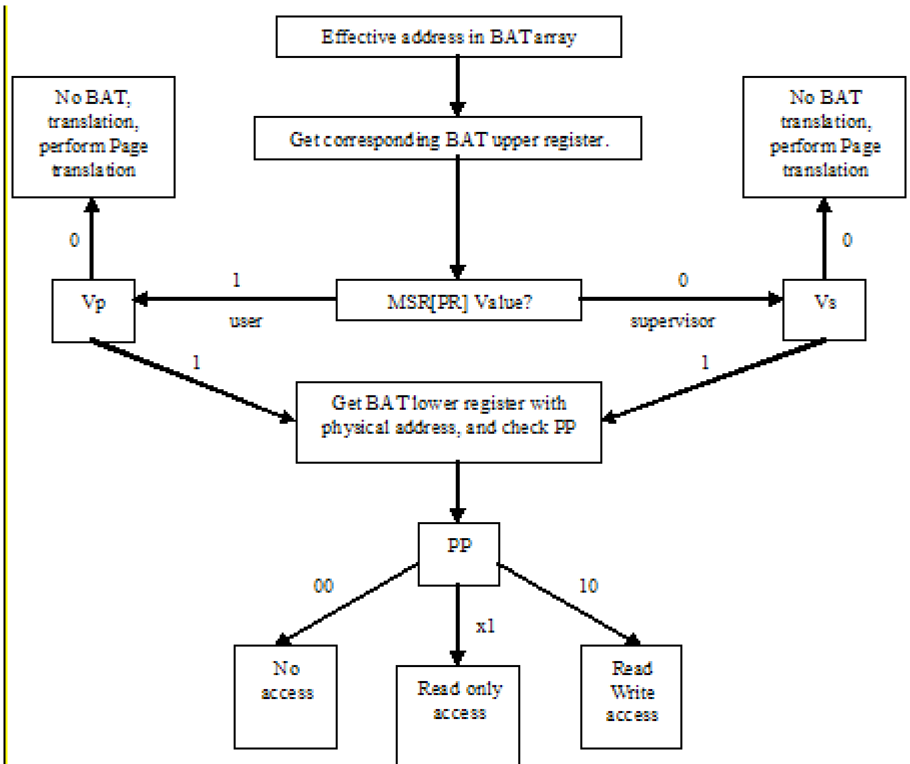


Figure 5-47: Block Address Translation access control

5.5.2.3.6 Page Address Translation and access control

If BAT is unable to perform address translation, Page Address Translation is used. Page Address Translation provides access control at the segment level and at the individual page level. Segment level access control allows the designation of a memory segment as data only. Page Address Translation mechanism prevents instructions from being fetched from these data only segments.

Page address translation begins with a check to see if the effective segment ID, corresponding to the effective address, exists in the Segment Lookaside Buffer (SLB). The SLB provides a mapping between Effective Segment Ids (ESIDs) and Virtual Segment Ids (VSIDs). If the SLB search fails, a segment fault occurs. This is an Instruction Segment exception or a data segment exception, depending on whether the effective address is for an instruction fetch or for a data access. The Segment Table Entry (STE) is then located with the Address Space Register and the segment table.

Page-level access control uses a key bit from Segment Table Entry (STE), along with the Page Protection (PP) bits from the Page Table Entry, to determine whether supervisor and user programs can access a page. Page access permissions are granular to no access, read only access, and read-write access. The key bit is calculated from the Machine State Register PR bit and Kp and Ks bits from the Segment Table Entry, as follows:

Key = (Kp & MSR[PR]) | (Ks & ~MSR[PR])

That is, in supervisor mode, use the Ks bit from the STE and ignore the Kp bit. In user mode, use the Kp bit and ignore the Ks bit.

The following diagram schematically describes the Page Address Translation mechanism and the access control performed by it.

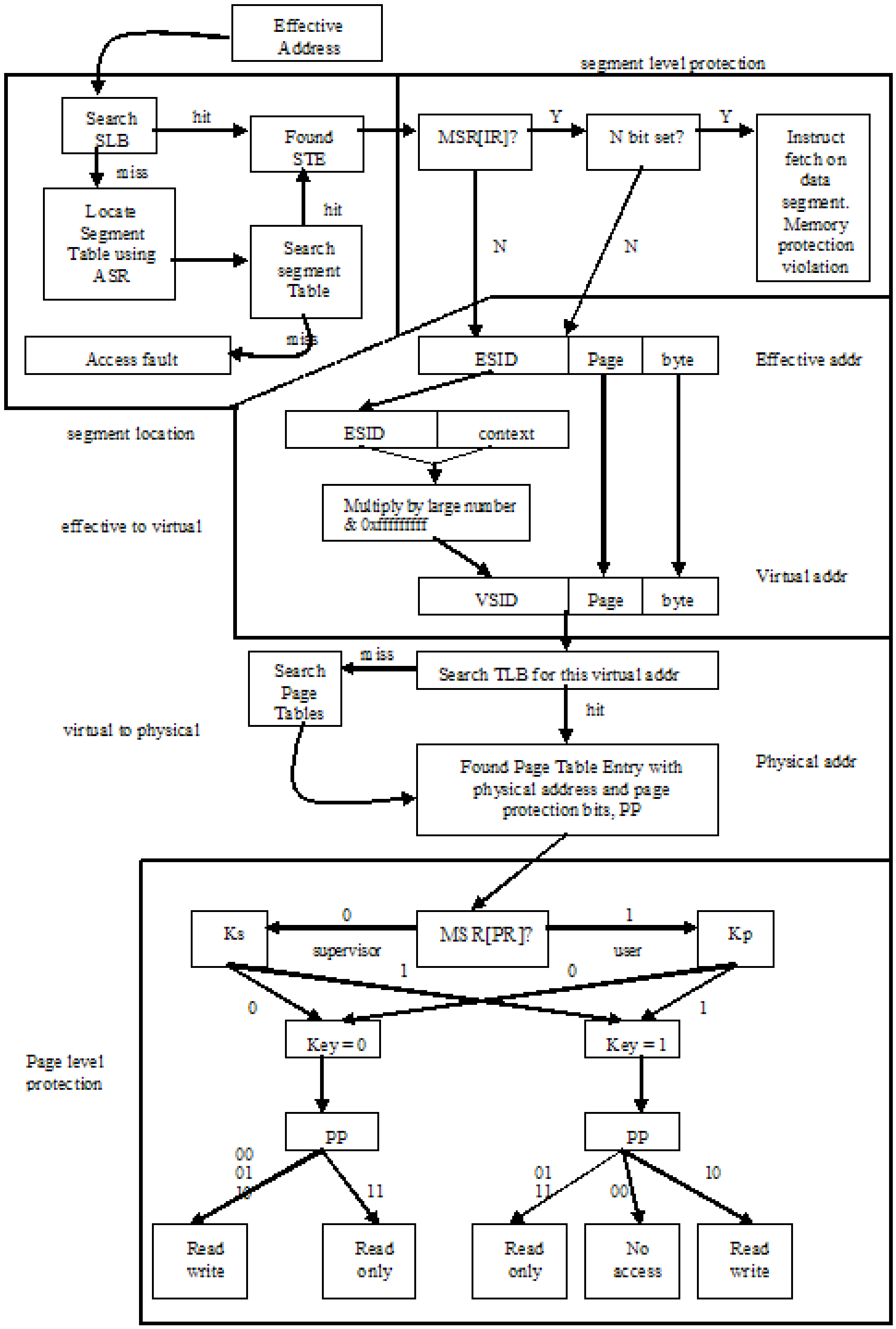


Figure 5-48: Page Address Translation and access control

5.5.2.4 System z

SLES on System z systems can run either in native mode or in LPAR. Additionally, it can run as z/VM guests, which is specific to this series. This section briefly describes these three modes and how they address and protect memory. For more detailed information about System z architecture, refer to *Z/Architecture Principle of Operation*, at http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf, or System z hardware documents at http://www-1.ibm.com/servers/eserver/zseries/.

5.5.2.4.1 Native hardware mode

SLES runs directly on System z hardware in native hardware mode. Only one instantiation of SLES can run at one time. All CPUs, memory, and devices are directly under the control of the SLES operating system. Native Hardware mode is useful when a single server requires a large amount of memory. Native Hardware mode is not very common, because it requires device driver support in SLES for all attached devices, and Native Hardware does not provide the flexibility of the other two modes.

5.5.2.4.2 LPAR mode

In LPAR mode, System z hardware is partitioned into up to thirty different partitions. The partitioned hardware is under the control of a hypervisor called the control program. Each partition is allocated a certain number of CPUs and a certain amount of memory. Devices can be dedicated to a particular partition, or they can be shared among several partitions. The control program provides preemptive time slicing between the partitions sharing a processor. It also ensures that each partition gets the exact allocated share of the CPU, even if the remainder of the processor is unused. SLES runs in one of these logical partitions. LPAR mode provides more flexibility than Native Hardware mode, but still requires device driver support for devices dedicated to a partition.

5.5.2.4.3 z/VM Guest mode

In z/VM Guest mode, SLES runs as a guest operating system on one or more z/VM virtual machines. z/VM virtualizes the hardware by providing the same interface definition provided by the real hardware to a guest operating system. Guests operate independently of each other, even though they share memory, processors, and devices. The z/VM Guest mode provides even more flexibility than LPAR mode because, unlike LPAR, z/VM virtual machines allow dynamic addition or deletion of memory and devices. The z/VM Guest mode is the most commonly deployed mode because of the flexibility that it provides.

In terms of memory addressing, all three modes believe they are operating directly on the System z hardware. The control program (either LPAR or VM, or both) sets up their paging tables and zoning array so the Start Interpretive Execution (SIE) instruction can do the address conversion. The control program does not actively convert any addresses.

5.5.2.4.4 Address types

z/Architecture defines four types of memory addresses: virtual, real, absolute, and effective. These memory addresses are distinguished on the basis of the transformations that are applied to the address during a memory access.

Virtual address: A virtual address identifies a location in virtual memory. When a virtual address is used to access main memory, it is translated by a Dynamic Address Translation (DAT) mechanism to a real address, which in turn is translated by prefixing to an absolute address. The absolute address of a virtualized system is, in turn, subjected to dynamic address translation in VM or to zoning in LPAR.

Real address: A real address identifies a location in real memory. When a real address is used to access main memory, it is converted by, prefixing mechanism, to an absolute address.

Absolute address: An absolute address is the address assigned to a main memory location. An absolute address is used for a memory access without any transformations performed on it.

Effective address: An effective address is the address that exists before any transformation takes place by dynamic address translation or prefixing. An effective address is the result of the address arithmetic of adding the base register, the index register, and the displacement. If DAT is on, the effective address is the same as the virtual address. If DAT is off, the effective address is the same as the real address.

5.5.2.4.5 Address sizes

z/Architecture supports 24-bit, 31-bit, and 64-bit virtual, real, and absolute addresses. Bits 31 and 32 of the Program Status Word (PSW) control the address size. If they are both zero, then the addressing mode is 24-bit. If they are 0 and 1, the addressing mode is 31-bit. If they are both 1, then the addressing mode is 64-bit. When addressing mode is 24-bit or 31-bit, 40 or 33 zeros, respectively, are appended on the left to form a 64-bit virtual address. The real address that is computed by dynamic address translation, and the absolute address that is then computed by prefixing, are always 64-bit.

5.5.2.4.6 Address spaces

An address space is a consecutive sequence of integer numbers, or virtual addresses, together with the specific transformation parameters, which allow each number to be associated with a byte location in memory. The sequence starts at zero and proceeds left to right. The z/Architecture provides the means to access different address spaces. In order to access these address spaces, there are four different addressing modes: primary, secondary, home, and access-register.

In the access-register mode, any number of address spaces can be addressed, limited only by the number of different Access List Entry Tokens (ALET) and the size of the main memory. The conceptual separation of kernel and user space of SLES is implemented using these address spaces. The kernel space corresponds to the primary address space, and the user space corresponds to the home address space. Access-register address space is used to implement a memory area that transfers data between kernel and user space. The secondary address space is not used in SLES. User programs, which run in the home-space translation mode, can only translate virtual addresses of the home address space. The separation protects the kernel memory resources from user-space programs.

5.5.2.4.7 Address translations

Address translation on z/Architecture can involve two steps. The first step, if dynamic address translation is turned on, involves the use of hierarchical page tables to convert a virtual address to a real address. The second step involves conversion of a real address to an absolute address, using prefixing. If dynamic address translation is turned off, the address translation consists of just one step, that of converting a real address to an absolute address.

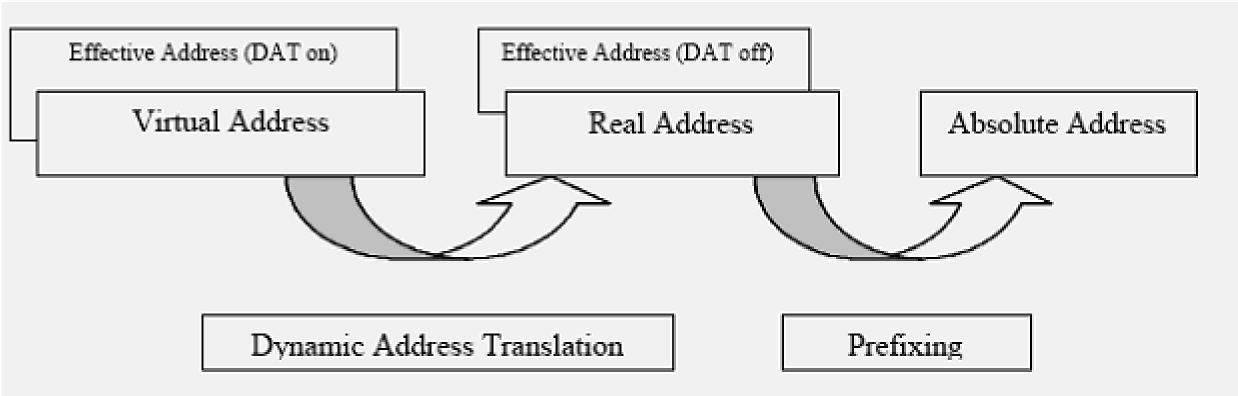


Figure 5-49: System z address types and their translation

5.5.2.4.7.1 Dynamic address translation

Bit 5 of the current PSW indicates whether a virtual address is to be translated using paging tables. If it is, bits 16 and 17 control which address space translation mode (primary, secondary, access-register, or home) is used for the translation.

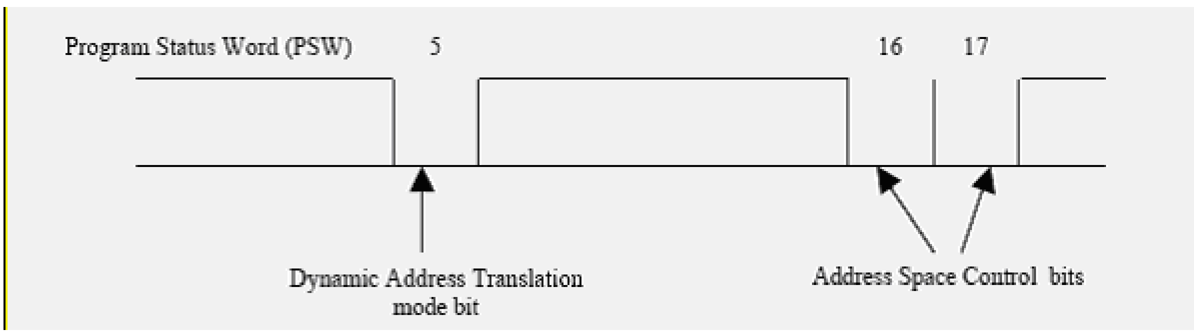


Figure 5-50: Program Status Word

The following diagram illustrates the logic used to determine the translation mode. If the DAT mode bit is not set, then the address is treated as a real address (Virtual = Real).

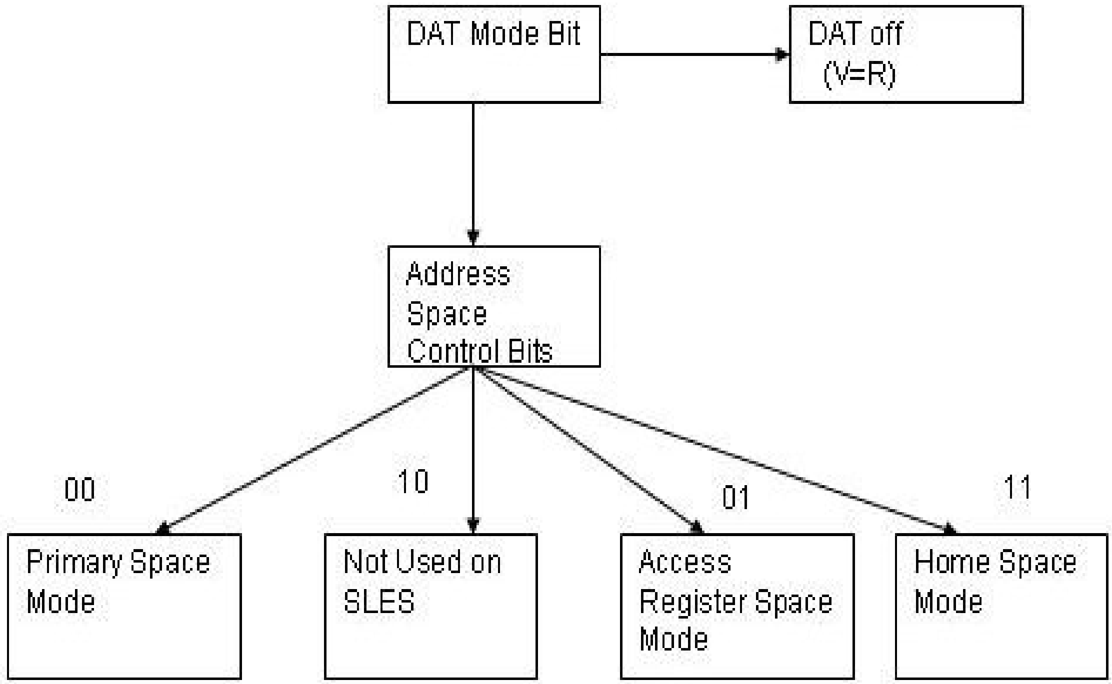


Figure 5-51: Address translation modes

Each address-space translation mode translates virtual addresses corresponding to that address space. For example, primary address-space mode translates virtual addresses from the primary address space, and home address space mode translates virtual addresses belonging to the home address space.

Each address space has an associated Address Space Control Element (ASCE). For primary address translation mode, the Primary Address Space Control Element (PASCE) is obtained from the CR1. For secondary address translation mode, the Secondary Address Space Control Element (SASCE) is obtained from the CR7. For home address translation mode, the Home Address Space Control Element (HASCE) is obtained from the CR13. In access-register translation mode, the Access List Entry Token (ALET) in the access register is checked. If it is the special ALET 0, PASCE is used. If it is the special ALET 1, SASCE is used. Otherwise, the ASCE found in the Address Space Number (ASN) table is used. SLES does not use the translation by the Address Space Number feature of the z/Architecture.

After the appropriate ASCE is selected, the translation process is the same for all of the four address translation modes. The ASCE of an address space contains the region table, for 64-bit addresses, or the segment table, for 31-bit addresses, origin. DAT uses that table-origin address to translate a virtual address to a real address, as illustrated below.

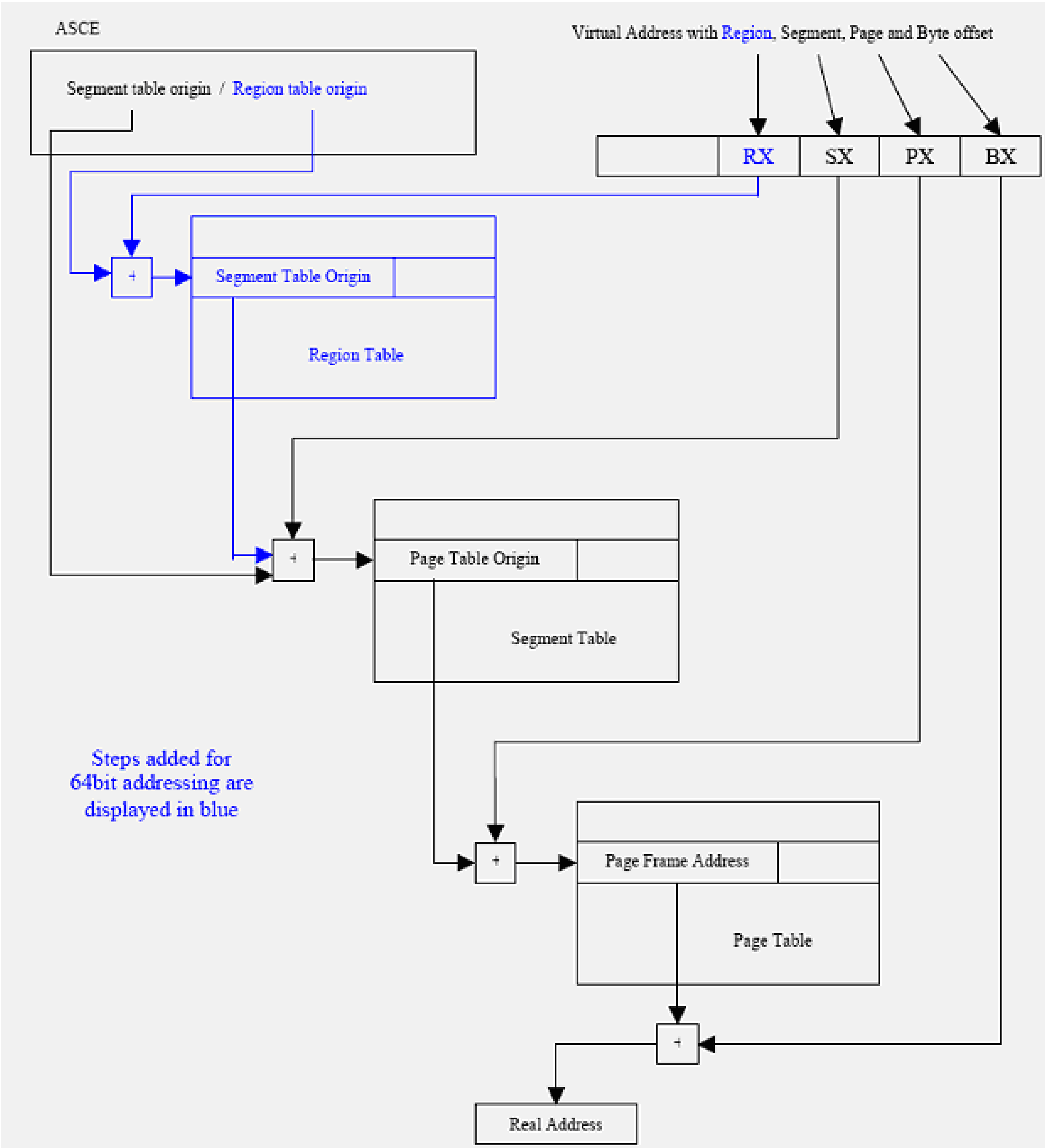


Figure 5-52: 64-bit or 31-bit Dynamic Address Translation

5.5.2.4.7.2 Prefixing

Prefixing provides the ability to assign a range of real addresses to a different block in absolute memory for each CPU, thus permitting more than one CPU sharing main memory to operate concurrently with a minimum of interference. Prefixing is performed with the help of a prefix register. No access control is performed while translating a real address to an absolute address.

For a detailed description of prefixing as well as implementation details, see *z/Architecture Principles of Operation* at http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf.

5.5.2.4.8 Memory protection mechanisms

In addition to separating the address space of user and supervisor states, the z/Architecture provides mechanisms to protect memory from unauthorized access. Memory protections are implemented using a combination of the PSW register, a set of sixteen control registers (CRs), and a set of sixteen access registers (ARs). The remainder of this section describes memory protection mechanisms and how they are implemented using the PSW, CRs, and ARs.

z/Architecture provides three mechanisms for protecting the contents of main memory from destruction or misuse by programs that contain errors or are unauthorized, low-address protection, page table protection, and key-controlled protection. The protection mechanisms are applied independently at different stages of address translation. Access to main memory is only permitted when none of the mechanisms prohibit access.

Low-address protection is applied to effective addresses, page table protection is applied to virtual addresses while they are being translated into real addresses, and key-controlled protection is applied to absolute addresses.

5.5.2.4.8.1 Low-address protection

The low-address protection mechanism provides protection against the destruction of main memory information used by the CPU during interrupt processing. This is implemented by preventing instructions from writing to addresses in the ranges 0 through 511 and 4096 through 4607 (the first 512 bytes of each of the first and second 4 KB address blocks).

Low-address protection is applied to effective addresses only if the following bit positions are set in control register 0 and the ASCE of the address space to which the effective address belongs.

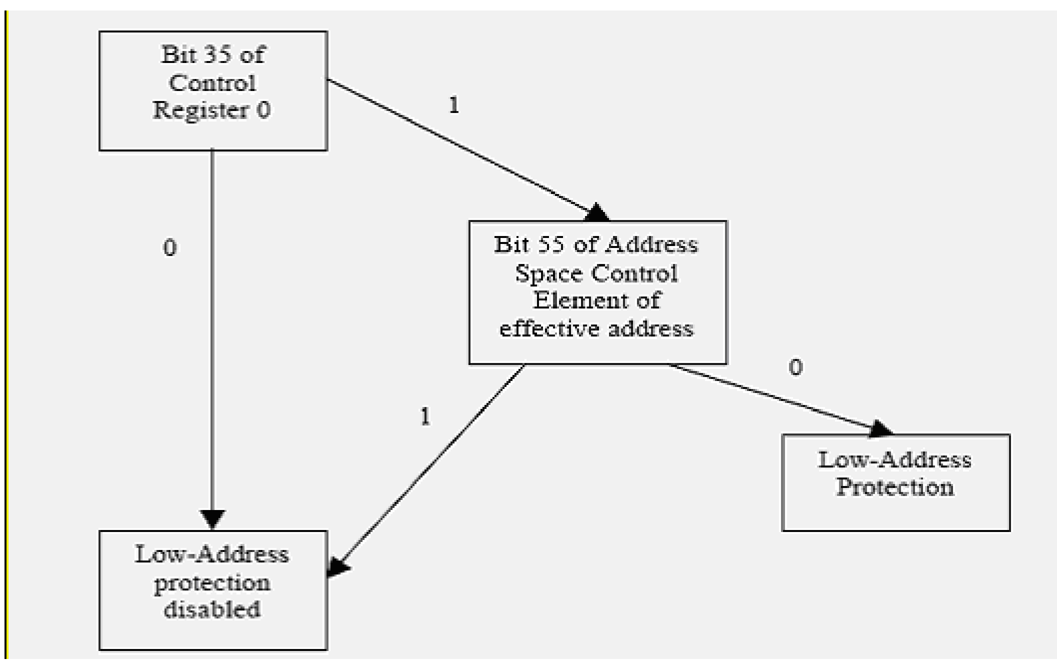


Figure 5-53: Low-address protection on effective address

5.5.2.4.8.2 Page table protection

The page table protection mechanism is applied to virtual addresses during their translation to real addresses. The page table protection mechanism controls access to virtual storage by using the page protection bit in each page-table entry and segment-table entry. Protection can be applied to a single page or an entire segment (a collection of contiguous pages). Once the ASCE is located, the following dynamic address translation is used to translate virtual address to a real address. Page table protection (for a page or a segment) is applied at this stage. The first diagram illustrates the DAT process for 31-bit addresses and the second diagram for the 64-bit addresses.

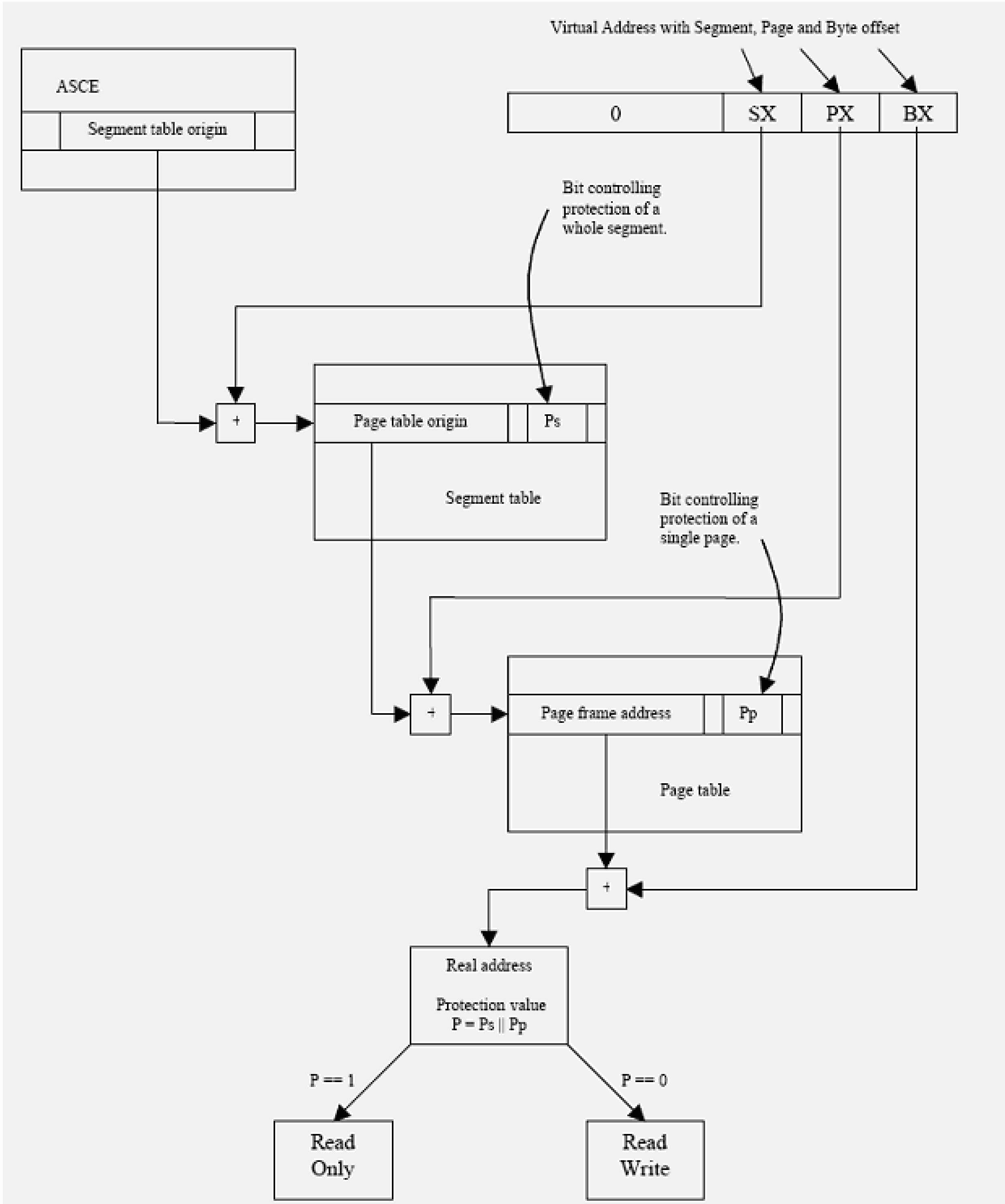


Figure 5-54: 31-bit Dynamic Address Translation with page table protection

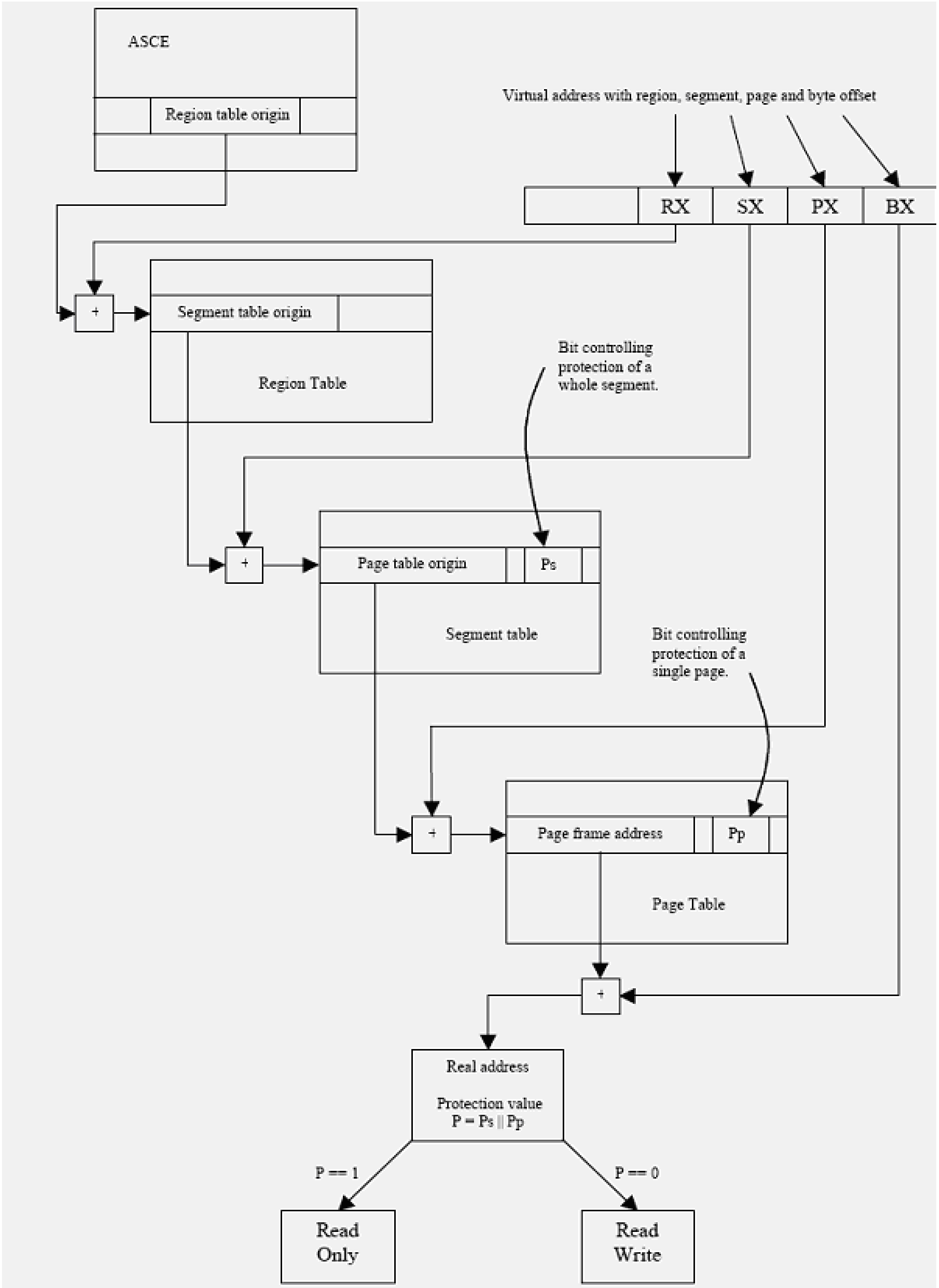


Figure 5-55: 64-bit Dynamic Address Translation with page table protection

5.5.2.4.8.3 Key-controlled protection

When an access attempt is made to an absolute address, which refers to a memory location, key-controlled protection is applied. Each 4K page, real memory location, has a 7-bit storage key associated with it. These storage keys for pages can only be set when the processor is in the supervisor state. The Program Status Word contains an access key corresponding to the current running program. Key-controlled protection is based on using the access key and the storage key to evaluate whether access to a specific memory location is granted.

The 7-bit storage key consists of access control bits (0, 1, 2, and 3), fetch protection bit (4), reference bit (5), and change bit (6). The following two diagrams show the key-controlled protection process.

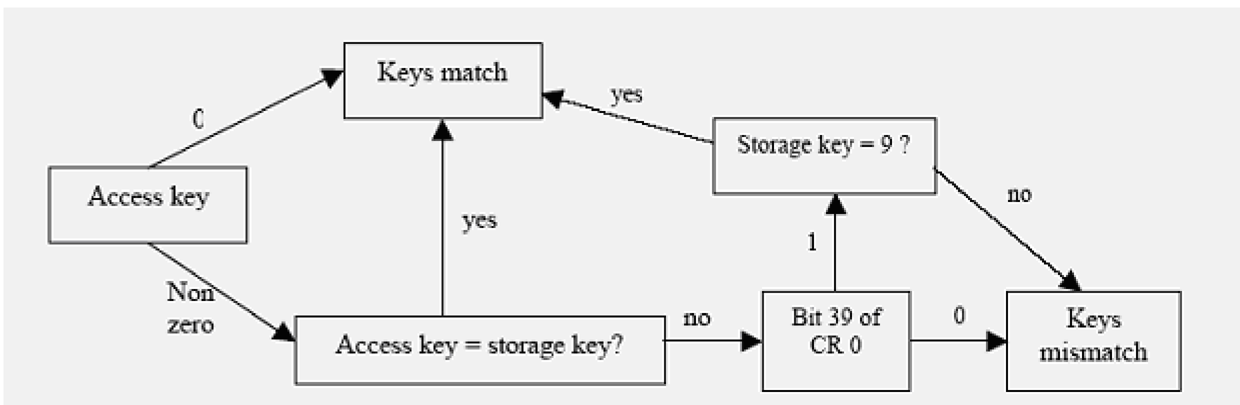


Figure 5-56: Key match logic for key-controlled protection

The z/Architecture allows for fetch protection override for key-controlled protection. The following diagram describes how fetch protection override can be used. Currently, SLES does not set the fetch protection bit of the storage key.

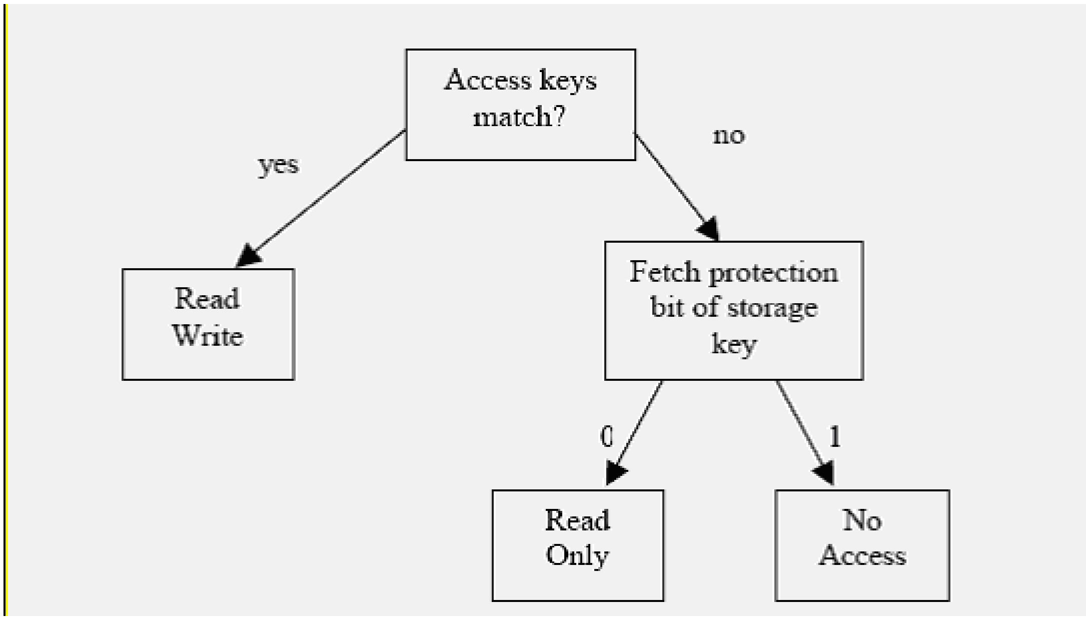


Figure 5-57: Fetch protection override for key-controlled

5.5.2.5 eServer 326

eServer 326 systems use AMD Opteron processors. The Opteron processors can either operate in legacy mode to support 32-bit operating systems, or in long mode to support 64-bit operating systems. Long mode has two possible sub modes, the 64-bit mode, which runs only 64-bit applications, and compatibility mode, which can run on both 32-bit and 64-bit applications simultaneously.

In legacy mode, the Opteron processor complies with the x86 architecture described in the System x sections of this document. SLES on eServer 326 uses the compatibility mode of the Opteron processor. The compatibility mode complies with x86-64 architecture, which is an extension of x86 architecture to support 64-bit applications along with legacy 32-bit applications. The following description corresponds to the x86-64 architecture.

This section briefly describes the eServer 326 memory addressing scheme. For more detailed information about the eServer 326 memory management subsystem, see *AMD64 Architecture, Programmer's Manual Volume 2: System Programming*, at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf and "Porting Linux to x86-64", by Andi Kleen, at http://old.lwn.net/2001/features/OLS/pdf/pdf/x86-64.pdf.

Access control and protection mechanisms are part of both segmentation and paging. The following sections describe the four address types on IBM eServer 326 computers (logical, effective, linear, and physical), and how segmentation and paging are used to provide access control and memory resource separation by SLES on IBM eServer 326 systems.

5.5.2.5.1 Logical address

A logical address is a reference into a segmented-address space. It is comprised of the segment selector and the effective address. Notationally, a logical address is represented as

Logical Address = Segment Selector: Offset

The segment selector specifies an entry in either the global or local descriptor table. The specified descriptor-table entry describes the segment location in virtual-address space, its size, and other characteristics. The effective address is used as an offset into the segment specified by the selector.

5.5.2.5.2 Effective address

The offset into a memory segment is referred to as an effective address. Effective addresses are formed by adding together elements comprising a base value, a scaled-index value, and a displacement value. The effective address computation is represented by the equation:

Effective Address = Base + (Scale x Index) + Displacement

Long mode supports 64-bit effective-address length.

5.5.2.5.3 Linear address

The linear address, which is also referred as virtual address, is a 64-bit address computed by adding the segment base address to the segment offset.

5.5.2.5.4 Physical address

A physical address is a reference into the physical-address space, typically main memory. Physical addresses are translated from virtual addresses using page translation mechanisms.

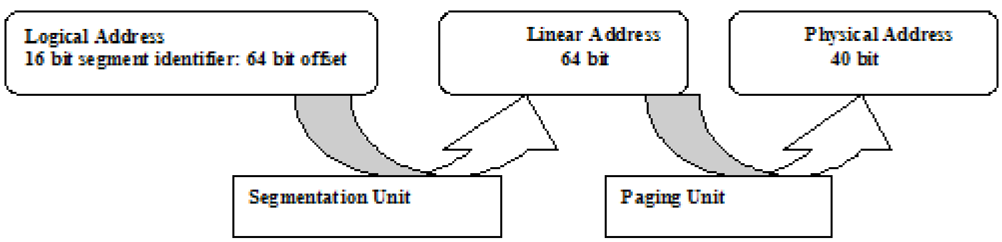


Figure 5-58: eServer 326 address types and their conversion units

5.5.2.5.5 Segmentation

Segmentation is a method by which system software can isolate software processes, or tasks, and the data used by those processes, from one another in an effort to increase the reliability of systems running multiple processes simultaneously. Segmentation is used both in compatibility mode and legacy mode.

The segment protection mechanism uses the concept of privilege levels that is similar to the one used by x86 architecture. The processor supports four different privilege levels with a numerical value from 0 to 3, with 0 being the most privileged, and 3 being the least privileged. SLES only needs two privilege levels, kernel and user, and implements them by assigning user level to privilege level 3, and kernel level to privilege levels 0, 1 and 2. The x86-64 architecture defines three types of privilege levels to control access to segments: current, requestor, and descriptor.

- Current Privilege Level (CPL): CPL is the privilege level at which the processor is currently executing. The CPL is stored in an internal processor register.

- Requestor Privilege Level (RPL):RPL represents the privilege level of the program that created the segment selector. The RPL is stored in the segment selector used to reference the segment descriptor.
- Descriptor Privilege Level (DPL):DPL is the privilege level that is associated with an individual segment. The system software assigns this DPL, and it is stored in the segment descriptor.

CPL, RPL, and DPL are used to implement access control on data accesses and control transfers as follows.

5.5.2.5.5.1 Access control for data access

When loading a data segment register, the processor checks privilege levels to determine if the load should succeed. The processor computes the subject's effective privilege as the higher numerical value, or lower privilege, between the CPL and the RPL. The effective privilege value is then compared with the object's privilege value, the DPL of the segment. Access is granted if the effective privilege value is lower than the DPL value (higher privilege). Otherwise, a general protection exception occurs, and the segment register is not loaded. The following diagram from [AMD64] illustrates data-access privilege checks.

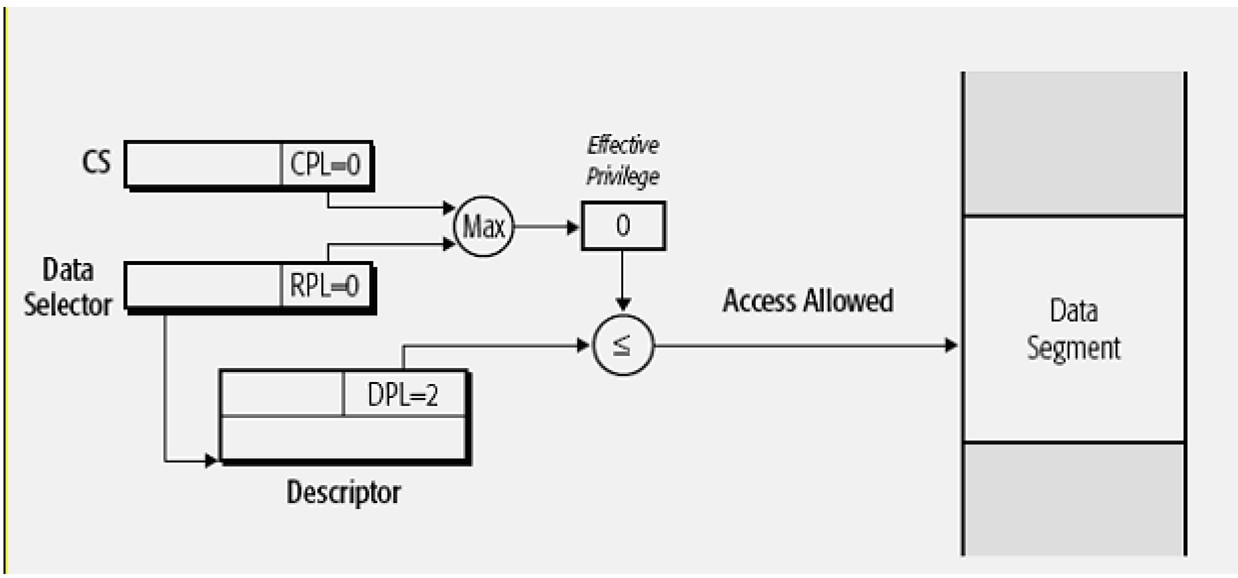


Figure 5-59: Data access privilege checks

5.5.2.5.5.2 Access control for stack segments

When loading a stack segment register, the processor ensures that the CPL and the stack selector RPL are equal. If they are not equal, a general protection exception occurs. If CPL and RPL are equal, then the processor compares the CPL with the DPL in the descriptor table entry that the segment selector references. If the two are equal, then the stack segment register is loaded. Otherwise, a general protection exception occurs, and the stack segment is not loaded.

5.5.2.5.5.3 Access control for direct control transfer

The processor performs privilege checks when control transfer is attempted between different code segments. Control transfer occurs with CALL/JMP instructions and SYSCALL/SYSRET instructions. Unlike the x86 architecture, the AMD Opteron provides the SYSCALL and SYSRET specific instructions to perform system calls. If the code segment is non-conforming (with conforming bit C set to zero in the segment descriptor), then the processor first checks to ensure that CPL is equal to DPL. If CPL is equal to DPL, then the processor performs the next check to see if the RPL value is less than or equal to the CPL. A general protection exception occurs if either of the two checks fail. If the code segment is conforming (with conforming bit C set to one in the segment descriptor), then the processor compares the target code-segment descriptor DPL with the currently executing program CPL. If the DPL is less than or equal to the CPL, then access is allowed. Otherwise, a general protection exception occurs. RPL is ignored for conforming segments.

5.5.2.5.5.4 Access control for control transfers through call gates

The AMD Opteron processor uses call gates for control transfers to higher privileged code segments. Call gates are descriptors that contain pointers to code-segment descriptors and control access to those descriptors. Operating systems can use call gates to establish secure entry points into system service routines. Before loading the code register with the code segment selector located in the call gate, the processor performs the following three privilege checks:

1. Compare the CPL with the call-gate DPL from the call-gate descriptor. The CPL must be less than or equal to the DPL.
2. Compare the RPL in the call-gate selector with the DPL. The RPL must be less than or equal to the DPL.
3. A call or jump through a call gate to a conforming segment requires that the CPL be greater than or equal to the DPL. Otherwise, a call or jump through a call gate requires that the CPL be equal to the DPL.

5.5.2.5.5.5 Access control through type check

After a segment descriptor is loaded into one of the segment registers, reads and writes into the segments are restricted based on type checks, as follows:

- Prohibit write operations into read-only data segment types.
- Prohibit write operations into executable code segment types.
- Prohibit read operations from code segments if the readable bit is cleared to 0.

5.5.2.5.6 Paging

The paging unit translates a linear address into a physical address. Linear addresses are grouped in fixed length intervals called pages. To allow the kernel to specify the physical address and access rights of a page instead of addresses and access rights of all the linear addresses in the page, continuous linear addresses within a page are mapped to continuous physical addresses.

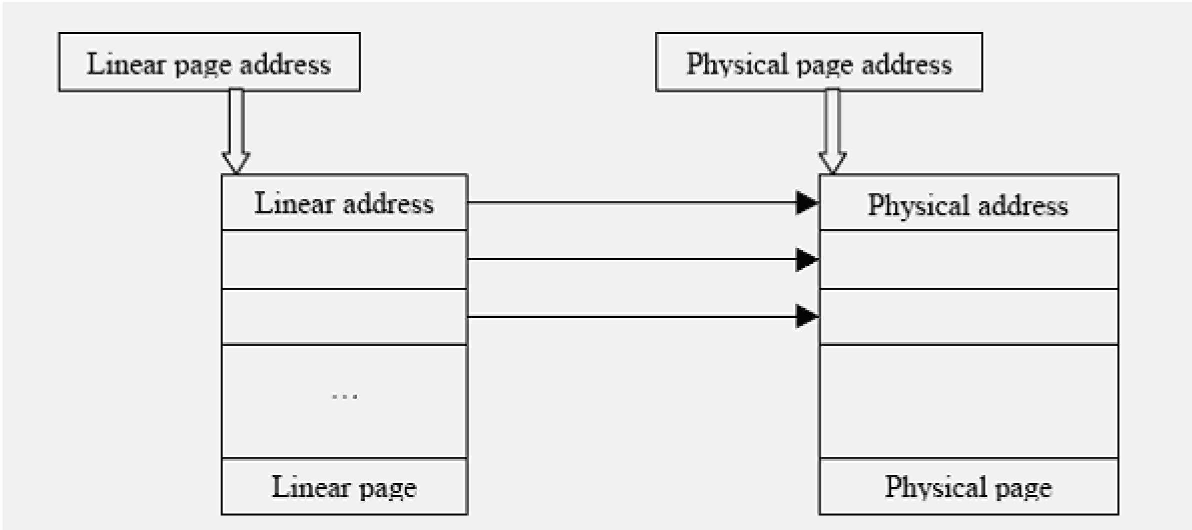


Figure 5-60: Contiguous linear addresses map to contiguous physical addresses

The eServer 326 supports a four-level page table. The uppermost level is kept private to the architecture-specific code of SLES. The page-table setup supports up to 48 bits of address space. The x86-64 architecture supports page sizes of 4 KB and 2 MB.

Figure 5-61 illustrates how paging is used to translate a 64-bit virtual address into a physical address for the 4 KB page size.

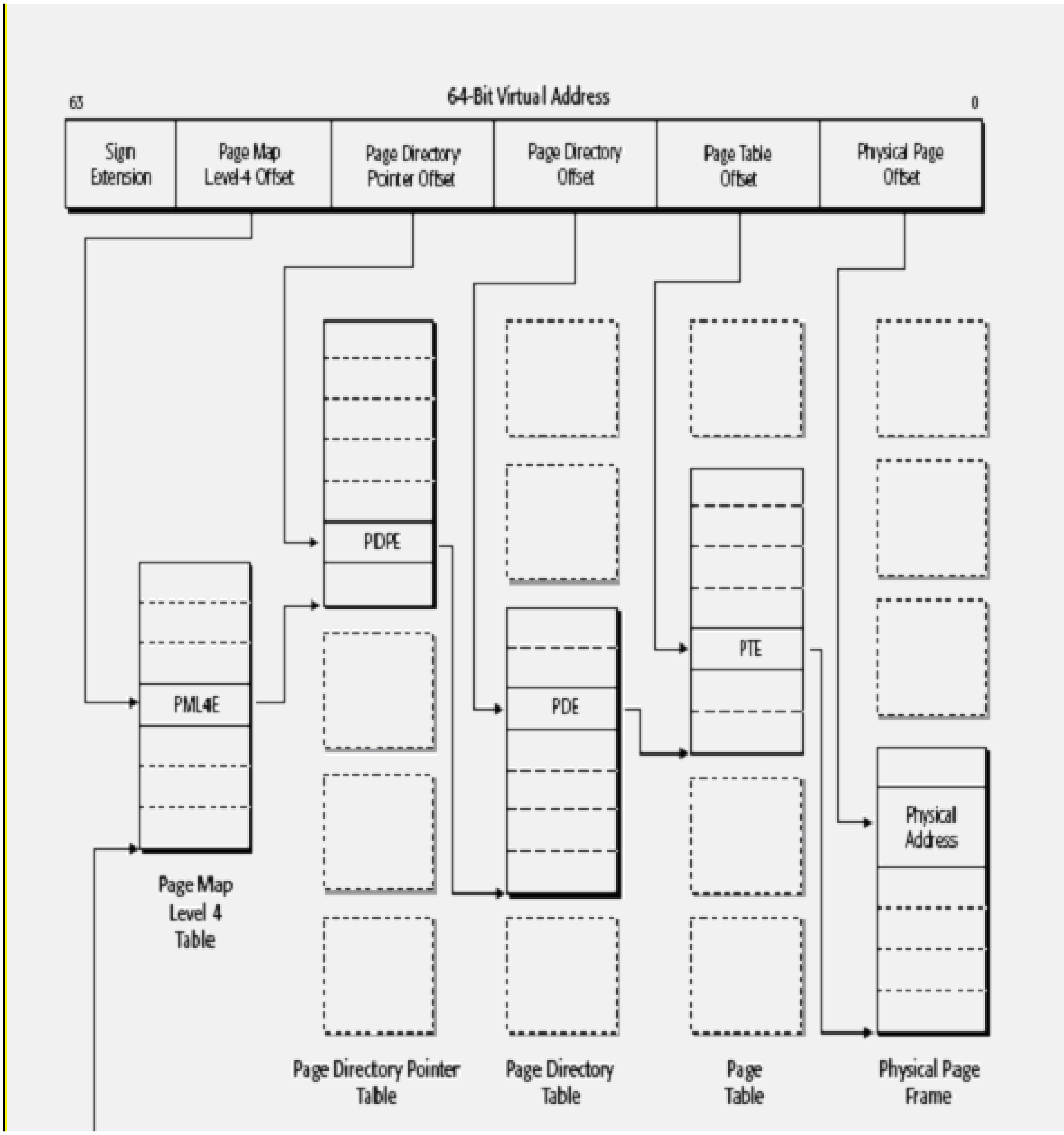


Figure 5-61: 4 KB page translation, virtual to physical address translation

When the page size is 2 MB, bits 0 to 20 represent the byte offset into the physical page. That is, page table offset and byte offset of the 4 KB page translation are combined to provide a byte offset into the 2 MB physical page. Figure 5-62 illustrates how paging is used to translate a 64-bit linear address into a physical address for the 2 MB page size.

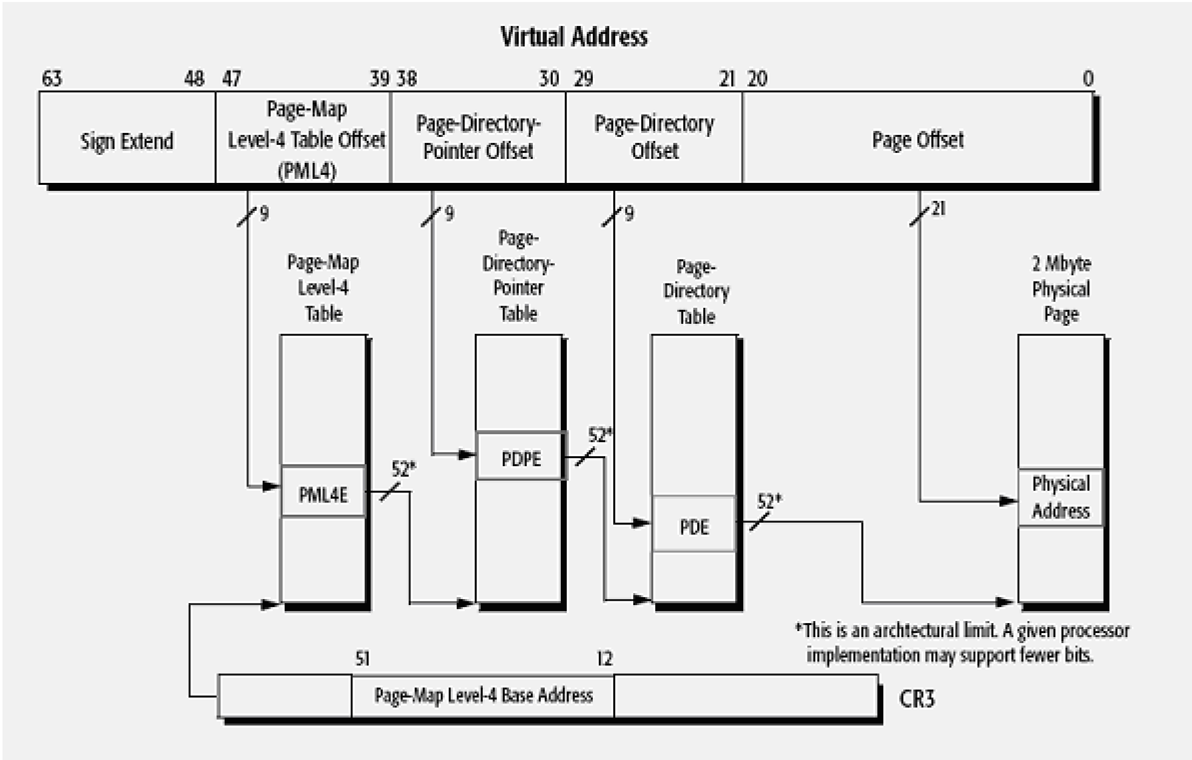


Figure 5-62: 2 MB page translation, virtual to physical address translation

Each entry of the page map level-4 table, the page-directory pointer table, the page-directory table, and the page table is represented by the same data structure. This data structure includes fields that interact in implementing access control during paging. These fields are the Read/Write (R/W) flag, the User/Supervisor (U/S) flag, and the No Execute (NX) flag.

The following diagram shows the bit positions in a page map level-4 entry. The flags hold the same bit positions for page directory pointer, page directory, page table, and page entries for both 4 KB page and 2 MB page sizes.

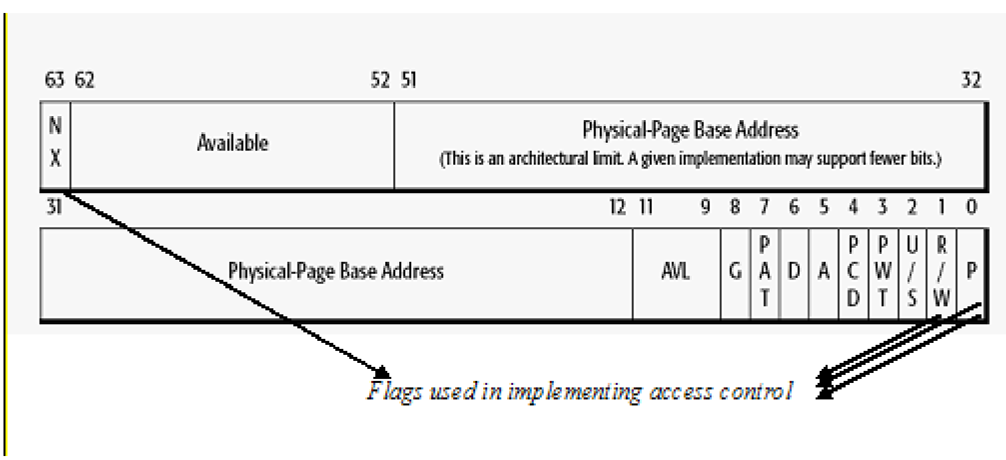


Figure 5-63: Page map level four entry

- Read/Write flag: This flag contains access rights of the physical pages mapped by the table entry. The R/W flag is either read/write or read. If set to 0, the corresponding page can only be read; otherwise, the corresponding page can be written to or read. The R/W flag affects all physical pages mapped by the table entry. That is, the R/W flag of the page map level-4 entry affects access to all the 128 MB (512 x 512 x 512) physical pages it maps through the lower-level translation tables.
- User/Supervisor flag: This flag controls the privilege level that is required to access the page or page table. The U/S flag is either 0, which indicates that the page can be accessed only in kernel mode, or 1, which indicates that it can always be accessed. This flag controls user access to all physical pages mapped by the table entry. That is, the U/S flag of the page map level-4 entry affects access to all the 128 MB (512 x 512 x 512) physical pages it maps through the lower-level translation tables.
- No Execute flag: This flag controls the ability to execute code from physical pages mapped by the table entry. When No Execute (NX) is set to 0, code can be executed from the mapped physical pages. Otherwise, when set to one, it prevents code from being executed from the mapped physical pages. This flag controls code execution from all physical pages mapped by the table entry. That is, the NX flag of the page map level-4 entry affects all 128 MB (512 x 512 x 512) physical pages it maps through the lower-level translation tables. The NX bit can only be set when the no-execute page-protection feature is enabled by setting the NXE bit of the Extended Feature Enable Register (EFER).

In addition to the R/W, U/S, and NX flags of the page entry, access control is also affected by the Write Protect (WP) bit of register CR0. If the write protection is not enabled (Write Protect bit set to 0), a process in kernel mode (CPL 0, 1 or 2) can write any physical page, even if it is marked as read-only. With write protection enabled, a process in kernel mode cannot write into read-only, user, or supervisor pages.

5.5.2.5.7 Translation Lookaside Buffers

The AMD Opteron processor includes an address translation cache called the Translation Lookaside Buffer (TLB) to expedite linear-to-physical address translation. The TLB is built up as the kernel performs linear to physical translations. Using the TLB, the kernel can quickly obtain a physical address corresponding to a linear address, without going through the page tables. Because address translations obtained from the TLB do not go through the paging access control mechanism described in Section 5.5.2.1.2, the kernel flushes the TLB buffer every time a process switch occurs between two regular processes. This process enforces the access control mechanism implemented by paging, as described in Section 5.5.2.1.2.

5.5.3 Kernel memory management

A portion of the RAM is permanently assigned to the SLES kernel. This memory stores kernel code and static data. The remaining part of RAM, called dynamic memory, is needed by the processes and the kernel itself.

Kernel memory management is highly improved in the SLES 2.6 kernel. Better memory management capabilities include support for Non Uniform Memory Access (NUMA) servers, Reverse map Virtual Memory (Rmap VM), huge TLBs, and Remap_file_pages. The following sections describe these improvements, and also describe page frame management, memory area management, and noncontiguous memory area management.

5.5.3.1 Support for NUMA servers

NUMA is an architecture wherein the memory access time for different regions of memory from a given processor varies according to the nodal distance of the memory region from the processor. Each region of memory, to which access times are the same from any CPU, is called a node. The NUMA architecture surpasses the scalability limits of SMP (Symmetric Multi-Processing) architecture.

With SMP, all memory accesses are posted to the same shared memory bus. This works fine for a relatively small number of CPUs, but there is a problem with hundreds of CPUs competing for access to this shared memory bus. NUMA alleviates these bottlenecks by limiting the number of CPUs on any one memory bus and connecting the various nodes by means of a high-speed interconnect. NUMA allows SLES Server to scale more efficiently for systems with dozens or hundreds of CPUs, because CPUs can access a dedicated memory bus for local memory. It also supports multiple interconnected memory nodes, each supporting a smaller number of CPUs.

Figure 5-64 shows a sample NUMA design. Each node in the system is simply a 4 processor SMP system. Each CPU in the node contains a L1 and L2 cache. The node contains an L3 cache, which is shared by all processors in the node. Nodes are connected together via an Interface/Node, which contains the SCI interface.

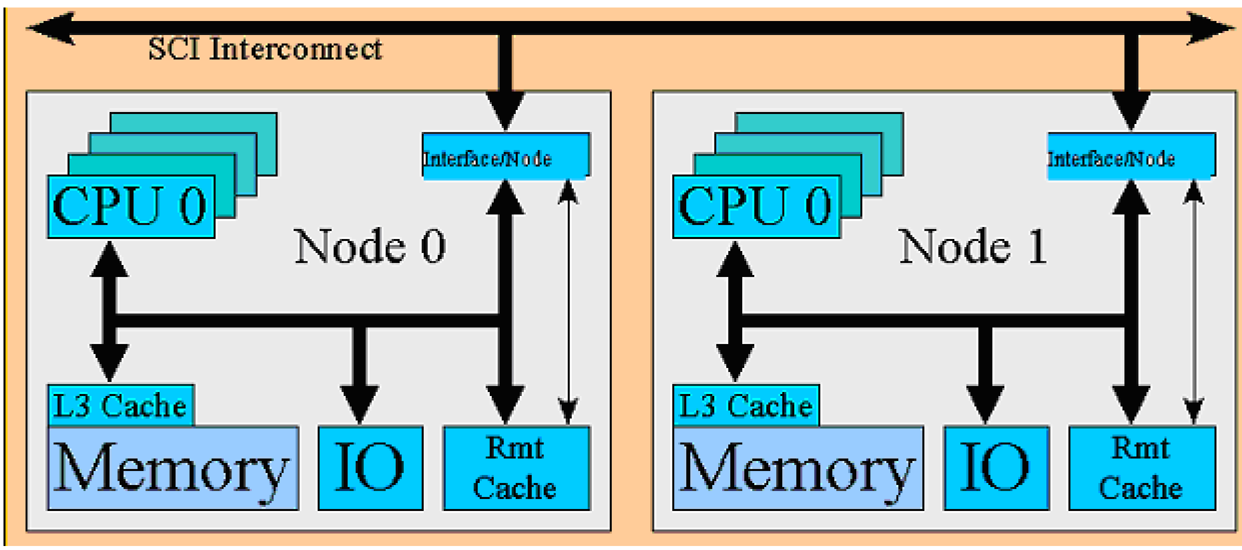


Figure 5-64: NUMA Design

The SLES kernel includes various data structures and macros for determining the layout of the memory and processors on the system. These enable the VM subsystem to make decisions on the optimal placement of memory for processes. For more information about NUMA, see http://lse.sourceforge.net/numa/.

5.5.3.2 Reverse map Virtual Memory

Reverse map Virtual Memory (Rmap VM) improves the operation of the kernel memory management subsystem, resulting in improved performance for memory constrained systems, NUMA systems, and systems with large aggregate virtual address spaces. Previously, the Linux memory management system could efficiently perform virtual-to-physical address translation using page tables. However, translating a physical address to its corresponding virtual addresses was inefficient and time consuming, because it required the memory management software to traverse every page table in the system. In today's large systems, this operation is unacceptably slow. With Rmap VM, additional memory management structures have been created that enable a physical address to be back-translated to its associated virtual address quickly and easily.

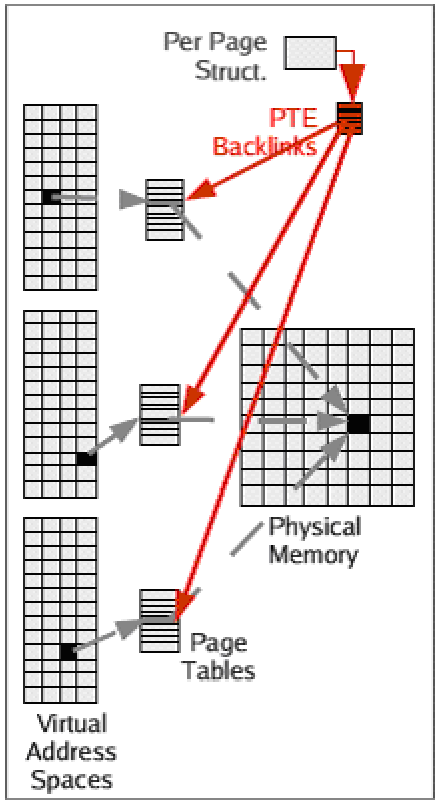


Figure 5-65: Rmap VM

For more information about Rmap VM, see http://lwn.net/Articles/23732/ and http://www-106.ibm.com/developerworks/linux/library/l-mem26/.

5.5.3.3 Huge Translation Lookaside Buffers

This memory management feature is valuable for applications that use a large virtual address space. It is especially useful for database applications. The CPU Translation Lookaside Buffer (TLB) is a small cache used for storing virtual-to-physical mapping information. By using the TLB, a translation can be performed without referencing the in-memory page table entry that maps the virtual address. However, to keep translations as fast as possible, the TLB is typically quite small, so it is not uncommon for large memory applications to exceed the mapping capacity of the TLB. The HugeTLB feature permits an application to use a much larger page size than normal, so a single TLB entry can map a correspondingly larger address space. A HugeTLB entry can vary in size, but, as an example, in an Itanium 2 system a huge page might be 1000 times larger than a normal page. This allows the TLB to map 1000 times the virtual address space of a normal process without incurring a TLB cache miss. For simplicity, this feature is exposed to applications by means of a file system interface.

Huge TLB File system (hugetlbfs) is a pseudo file system, implemented in fs/hugetlbfs/inode.c. The basic idea behind the implementation is that large pages are being used to back up any file that exists in the file system.

During initialization, init_hugetlbfs_fs() registers the file system and mounts it as an internal file system with kern_mount().

There are two ways that a process can access huge pages. The first is by using shmget() to set up a shared region backed by huge pages, and the second is the mmap() call on a file opened in the huge page file system.

When a shared memory region should be backed by huge pages, the process should call shmget() and pass SHM_HUGETLB as one of the flags. This results in hugetlb_zero_setup() being called, which creates a new file in the root of the internal hugetlbfs. A file is created in the root of the internal file system. The name of the file is determined by an atomic counter called hugetlbfs_counter, which is incremented every time a shared region is set up.

To create a file backed by huge pages, the system administrator must first mount a hugetlbfs type of file system. Instructions on how to perform this task are detailed in Documentation/vm/hugetlbpage.txt on SLES systems. After the file system is mounted, files can be created as normal with the system call open(). When mmap() is called on the open file, the file_operations struct hugetlbfs_file_operations ensures that hugetlbfs_file_ mmap() is called to set up the region properly.

Huge TLB pages have their own function for the management of page tables, address space operations, and file system operations. The names of the functions for page table management can all be seen in linux/hugetlb.h, and they are named very similarly to their normal page equivalents. The implementation of the hugetlbfs functions are located near their normal page equivalents, so they are easy to find.

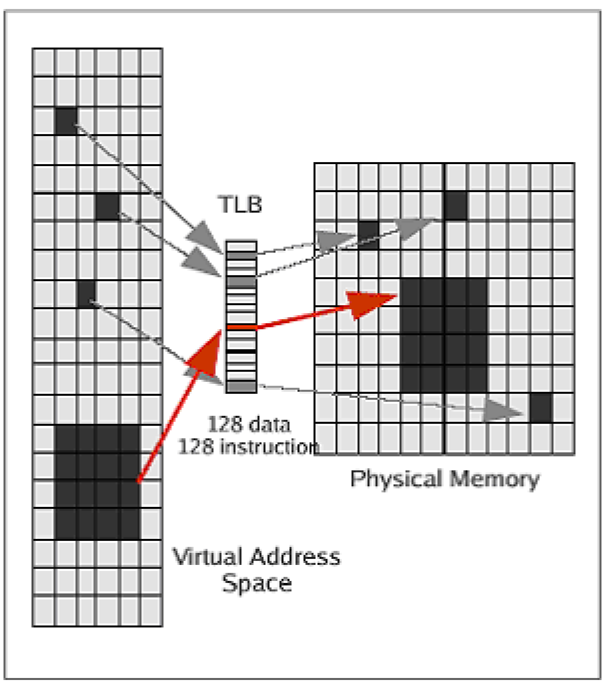


Figure 5-66: TLB Operation

5.5.3.4 Remap_file_pages

Remap_file_pages is another memory management feature that is suitable for large memory and database applications. It is primarily useful for x86 systems that use the shared memory file system (`shmemfs`). A `shmemfs` memory segment requires kernel structures for control and mapping functions, and these structures can grow unacceptably large given a large enough segment and multiple sharers.

For example, a 512 MB segment requires approximately 1 MB of kernel mapping structures per accessing process. Large database applications that create hundreds of concurrent threads (one for each SQL query, for example) can quickly consume all available free kernel address space. The Remap_file_pages feature modifies the shmemfs management structures, so they are significantly smaller (less than 100 bytes). This permits much larger shmemfs segments, and thousands of concurrent users to be supported.

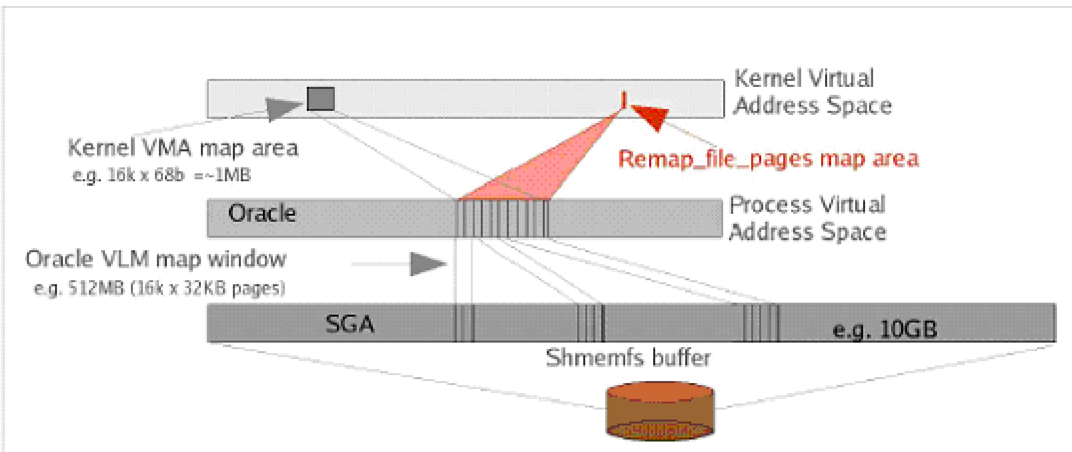


Figure 5-67: Remap_file_pages for database applications

This section describes dynamic memory used by the kernel, and describes the object reuse requirement. This section also discusses the three sections of kernel memory management: Page Frame Management, Memory Area Management, and Noncontiguous Memory Area Management.

5.5.3.5 Page frame management

The fundamental unit of memory under Linux is the page, a non-overlapping region of contiguous memory. SLES uses 4 KB pages for most processors as the standard memory allocation unit. The kernel keeps track of the current status of each page frame and distinguishes the page frames that are used to contain pages that belong to processes from those that contain kernel code and data. Page frames that are to be used by processes are allocated with the `get_zeroed_page()` routine. The routine invokes the function `alloc_pages()`. The routine then fills the page frame it obtained with zeros by calling `clear_page()`, thus satisfying the object reuse requirement.

5.5.3.6 Memory area management

Memory areas are sequences of memory cells having contiguous physical addresses with an arbitrary length. The SLES kernel uses the buddy algorithm for dealing with relatively large memory requests, but in order to satisfy kernel needs of small memory areas, a different scheme, called slab allocator, is used. The slab allocator views memory areas as objects with data and methods. Most of the kernel functions tend to repeatedly request objects of the same type, such as process descriptors, file descriptors, and so on. Hence, the slab allocator does not discard objects, but caches them so that they can be reused.

The slab allocator interfaces with the page frame allocator algorithm, Buddy System, to obtain free contiguous memory. The slab allocator calls the `kmem_getpages()` function with a flag parameter that indicates how the page frame is requested. This flag diverts the call to `get_zeroed_page()` if the memory area is to be used for a user mode process. As noted before, `get_zeroed_page()` initializes the newly allocated memory area with zero, thus satisfying the object reuse requirement.

5.5.3.7 Noncontiguous memory area management

In order to satisfy infrequent memory requests, SLES kernel uses noncontiguous memory area allocation methods which will reduce external fragmentation. The SLES kernel provides a mechanism via the `vmalloc()` function where non-contiguous physically memory can be used that is contiguous in virtual memory.

To allocate memory for kernel use, `vmalloc()` calls `__vmalloc()` with a `gfp_mask` flag that is always set to `GFP_KERNEL __GFP_HIGHMEM`. `__vmalloc()` in turn calls `vmalloc_area_pages()`, which will allocate the PTEs for the page requested.

5.5.4 Process address space

The address space of a process consists of all logical addresses that the process is allowed to use. Each process has its own address space, unless it is shared. The kernel allocates logical addresses to a process in intervals called memory regions. Memory regions have an initial logical address, a length, and some access rights.

This section highlights how the SLES kernel enforces separation of address spaces belonging to different processes using memory regions. It also highlights how the kernel prevents unauthorized disclosure of information by handling object reuse for newly allocated memory regions.

Some of the typical situations in which a process gets new memory regions are:

- creating a new process (`fork()`)
- loading an entirely new program (`execve()`)
- memory mapping a file (`mmap()`)
- creating shared memory (`shmat()`)
- expanding its heap (`malloc()`) and for growing its stack

All information related to the process address space is included in the memory descriptor (`mm_struct`) referenced by the mm field of the process descriptor. Memory descriptors are allocated from the slab allocator cache using `mm_alloc()`. Each memory region, represented by the `vm_area_struct` structure, identifies a linear address interval. Memory regions owned by a process never overlap.

To grow or shrink a process's address space, the kernel uses the `do_mmap()` and `do_unmap()` functions. The `do_mmap()` function calls `arch_get_unmapped_area()` to find an available linear address interval. Because linear address intervals in memory regions do not overlap, it is not possible for the linear address returned by `arch_get_unmapped_area()` to contain a linear address that is part of another process's address space.

In addition to this process compartmentalization, the `do_mmap()` routine also makes sure that when a new memory region is inserted it does not cause the size of the process address space to exceed the threshold set by the system parameter `rlimit`. The `do_mmap()` function only allocates a new valid linear address to the address space of a process. Actual page-frame allocation is deferred until the process attempts to access that address for a write operation. This technique is called demand paging. When accessing the address for a read operation, the kernel gives the address an existing page called Zero Page, which is filled with zeros. When accessing the address for a write operation, the kernel invokes the `alloc_page()` routine and fills the new page frame with zeros by callig `memset()`, thus satisfying the object reuse requirement.

The following diagram describes a simplified view of what occurs when a process tries to increase its address space and, if successful, tries to access the newly allocated linear address.

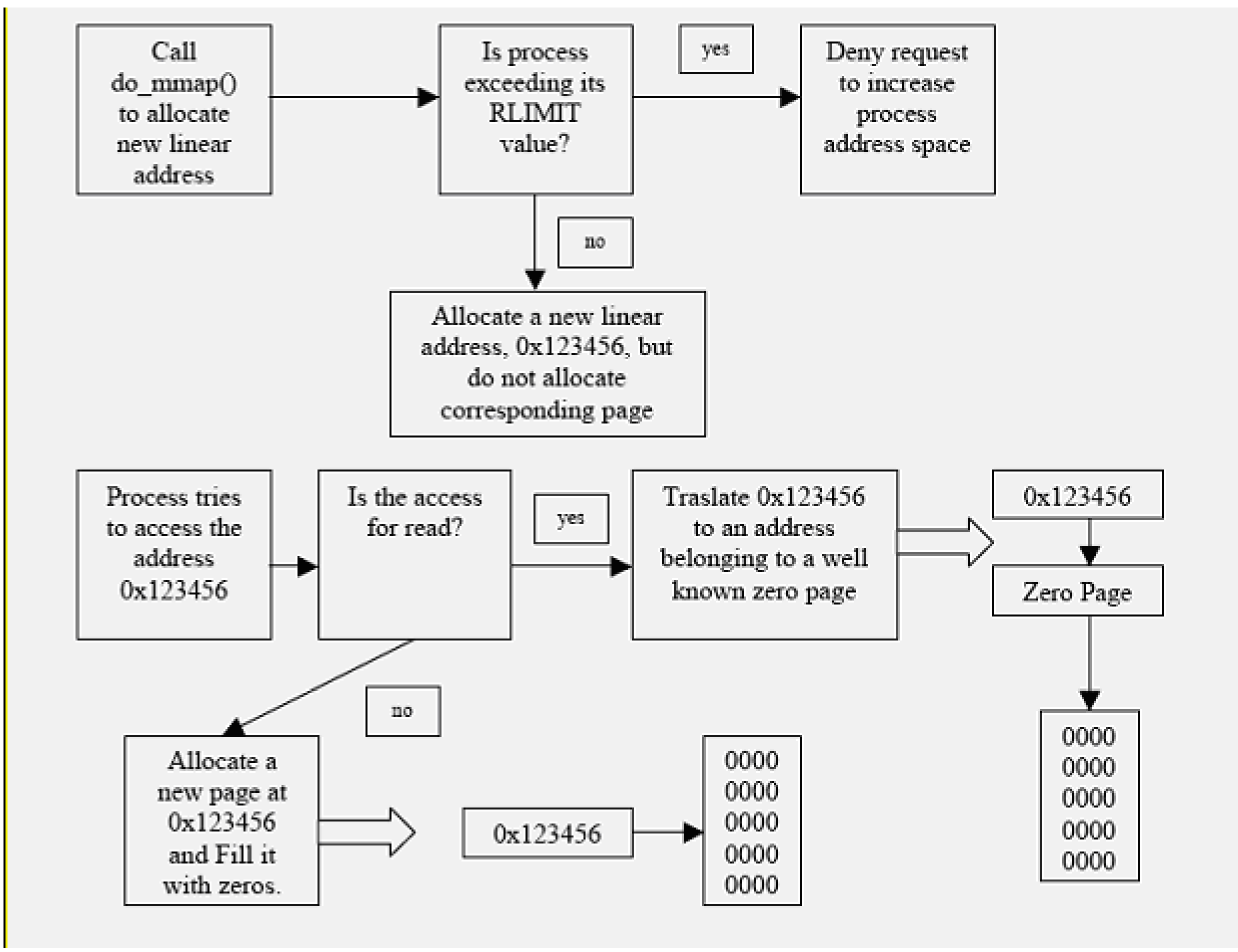


Figure 5-68: Object reuse handling while allocating new linear address

5.5.5 Symmetric multiprocessing and synchronization

The SLES kernel allows multiple processes to execute in the kernel simultaneously (the kernel is reentrant). It also supports symmetric multiprocessing (SMP), in which two or more processors share the same memory and have equal access to I/O devices. Because of re-entrancy and SMP synchronization, issues arises. This section describes various synchronization techniques used by the SLES kernel.

A kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt. If a kernel control path is suspended while acting on a kernel data structure, no other kernel control path should be allowed to act on the same data structure. A critical region is any section of code that must be completely executed by any kernel control path that enters it before another kernel control path can enter it. A critical region could be as small as code to update a global variable, or larger multi-instruction code to remove an element from a linked list. Depending on the size of, or the type of, operation performed by a critical region, the SLES kernel uses the following methods to implement synchronization.

5.5.5.1 Atomic operations

Operations that are completed without any interruption are called atomic operations. They constitute one of the synchronization techniques of Linux, and operate at the chip level. The operation is executed in a single instruction without being interrupted in the middle and avoids accesses to the same memory location by other CPUs. The SLES kernel provides a special `atomic_t` data type and special functions that act on `atomic_t` variables.

The compiler for the System x processor generates assembly language code with a special lock byte (0xf0) around instructions involving `atomic_t` type variables and functions. When executing these assembly instructions, the presence of the lock byte causes the control unit to lock the memory bus, preventing other processors from accessing the same memory location.

5.5.5.2 Memory barriers

Compilers and assemblers reorder instructions to optimize code. A memory barrier ensures that all instructions before the memory barrier are completed, before starting the instructions after the memory barrier. This is to ensure that compilers do not reorder instructions when dealing with critical sections. All synchronization primitives act as memory barriers (firewall).

The eServer series processors provide the following kinds of assembly language instructions that act as memory barriers:

- Instructions that operate on I/O ports.
- Instructions prefixed by a lock byte.
- Instructions that write into control registers, system registers, or debug registers.
- Special assembly languages instructions. An example is `iret`.
- Memory barrier primitives, such as `mb()`, `rmb()`, and `wmb()`, which provide memory barriers, read memory barriers, and write memory barriers, respectively, for multiprocessor or uniprocessor systems.
- Memory barrier primitives, such as `smp_mb()`, `smp_rmb()`, and `smp_wmb()`, which provide memory barriers, read memory barriers, and write memory barriers, respectively, for multiprocessor systems only.

5.5.5.3 Spin locks

Spin locks provide an additional synchronization primitive for applications running on SMP systems. A spin lock is just a simple flag. When a kernel control path tries to claim a spin lock, it first checks whether or not the flag is already set. If not, then the flag is set, and the operation succeeds immediately. If it is not possible to claim a spin lock, then the current thread spins in a tight loop, repeatedly checking the flag until it is clear. The spinning prevents more than one thread of execution from entering the critical region at any one time.

5.5.5.4 Kernel semaphores

A kernel semaphore is also a simple flag. If a kernel control path can acquire the semaphore, because the flag is not set, it proceeds with its normal operation. Otherwise, the corresponding process is suspended. It becomes runnable again once the semaphore is available. Semaphores are used to protect critical regions of code or data structures that allow just one process at a time to access critical regions of code and data; all other processes wishing to access this resource will be made to wait until it becomes free. The Linux kernel from version 2.6 provides mutex abstraction layer. For additional information on kernel mutexes, please see "Mutex Conundrum in the Linux Kernel," http://developer.osdl.org/dev/mutexes/docs/MutexSIG.pdf.

5.6 Audit subsystem

An auditing facility records information about actions that may affect the security of a computer system. In particular, an auditing facility records any action by any user that may represent a breach of system security. For each action, the auditing facility records enough information about those actions to verify the following:

- the user who performed the action
- the kernel object on which the action was performed
- the exact date and time it was performed
- the success or failure of the action
- the identity of the object involved

The TOE includes a comprehensive audit framework called Linux Audit Framework (LAF), which is composed of user-space and kernel-space components. The framework records security events in the form of an audit trail and provides tools to an administrative user to configure the subsystem and search for particular audit records, providing the administrator with the ability to identify attempted and realized violations of the system's security policy.

This section describes the design and operation at a high level.

5.6.1 Audit components

Figure 5-69 illustrates the various components that make up the audit framework and how they interact with each other. In general, there are user-space components and kernel-space components that use a netlink socket for communication. Whenever a security event of interest occurs, the kernel queues a record describing the event and its result to the netlink socket. If listening to the netlink, the audit daemon, `auditd`, reads the record and writes it to the audit log.

This section describes the various components of the audit subsystem, starting with the kernel components and then followed by the user-level components.

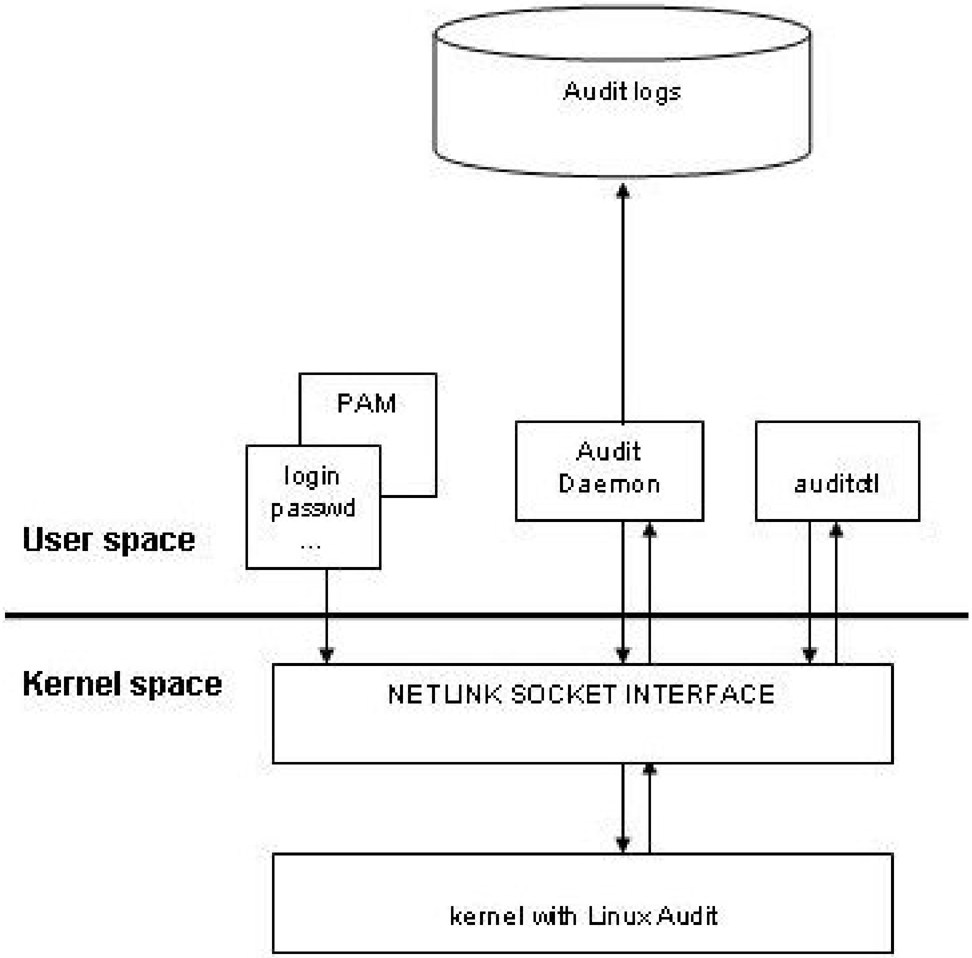


Figure 5-69: Audit framework components

5.6.1.1 Audit kernel components

Linux Audit of the SLES kernel includes three kernel-side components relating to the audit functionality. The first component is a generic mechanism for creating audit records and communicating with user space. The communication is achieved via netlink socket interface. Netlink enables the transfer of information between kernel modules and user-space processes. It provides kernel-user space bidirectional communication links. Linux Audit consists of a standard sockets-based interface for user processes and an internal kernel API for kernel modules.

5.6.1.1.1 Kernel-userspace interface

On top of netlink, there exists the generic netlink family that provides simplified access for less demanding users. This introduces a control for ID management and name resolution, and possesses a new type of safety interface for netlink messages and attributes handling. This interface also features simplified message constructing, validation capabilities, and documentation.

This first component also receives user-space commands to control the operation of the audit framework and to set the audit filter rules and file system watch points.

The kernel checks the effective capabilities of the sender process. If the sender does not possess the right capability, the netlink message is discarded.

5.6.1.1.2 Syscall auditing

The second component is a mechanism that addresses system call auditing. It uses the generic logging mechanism for creating audit records and communicating with user space.

5.6.1.1.3 Filesystem watches

The third component adds file system auditing support, based on file locations and names, to the audit subsystem. It employs the kernel inotify mechanism (Section 5.1.4) to track of all the objects being watched and a hash table to store the audit information for the inodes of the objects. Inotify is used to keep the rules list up to date. The audit filtering for a filesystem object is done based on dev and/or inode depending on whether dev/inode is specified in the rule or whether a path is specified.

At kernel startup four lists are created to hold the filter rules. One list is checked at task creation, another is checked at syscall entry time, the third is checked at syscall exit time, and the fourth is used to filter user messages. These lists hold the filter rules set by user-space components. Multiple variables are used to control the operation of audit.

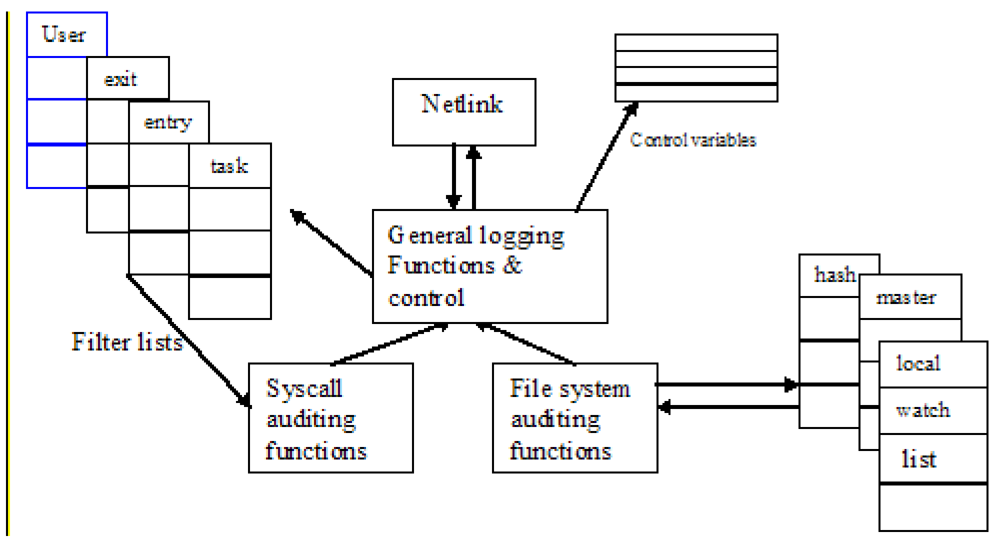


Figure 5-70: Audit Kernel Components

5.6.1.1.4 Task structure

The Audit Subsystem extends the task structure to potentially include an audit context. By default, on task creation, the audit context is built, unless specifically denied by the per-task filter rules. Then, during system calls the audit context data is filled. The audit subsystem further extends the audit context to allow for more auxiliary audit information, which might be needed for specific audit events.

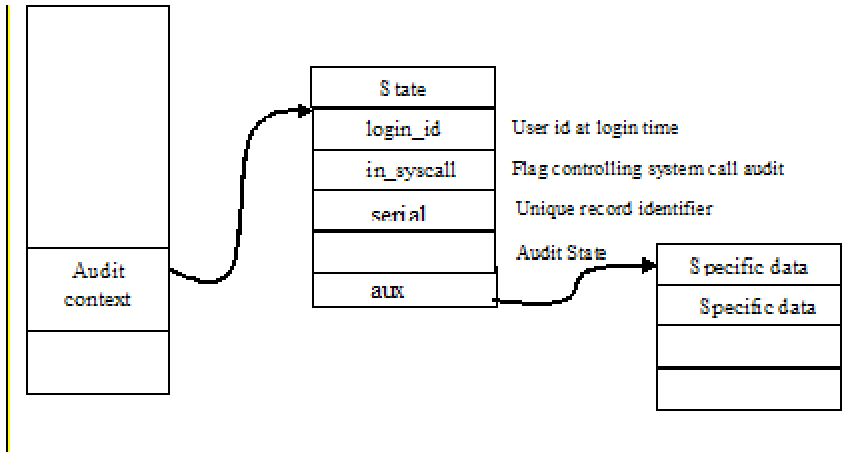


Figure 5-71: Task Structure

5.6.1.1.5 Audit context fields

- Login ID: Login ID is the user ID of the logged-in user. It remains unchanged through the `setuid()` or `seteuid()` system calls. Login ID is required by the Controlled Access Protection Profile to irrefutably associate a user with that user's actions, even across `su()` calls or use of setuid binaries.
- `state`: `state` represents the audit state that controls the creation of per-task audit context and filling of system call specifics in the audit context. It can take the following values:

`AUDIT_DISABLED`	Do not create per-task audit_context. No syscall specific audit records will be generated for the task
`AUDIT_SETUP_CONTEXT`	Create the per task `audit_context`, but don't necessarily fill it in a syscall entry time (i.e., filter instead).
`AUDIT_BUILD_CONTEXT`	Create the per task `audit_context`, and always fill it in at syscall entry time. This makes a full syscall record available if some other part of the kernel decides it should be recorded.
`AUDIT_RECORD_CONTEXT`	Create the per task `audit_context`, always fill it in at syscall entry time, and always write out the audit record at syscall exit time.

Table 5-1: Audit Context States

- `in_syscall`: States whether the process is running in a syscall versus in an interrupt.

- serial: A unique number that helps identify a particular audit record. Along with ctime, it can determine which pieces belong to the same audit record. The (timestamp, serial) tuple is unique for each syscall and it lives from syscall entry to syscall exit.
- ctime: Time at system call entry.
- major: System call number.
- argv array: The first 4 arguments of the system call.
- name_count: Number of names. The maximum defined is 20.
- audit_names: An array of audit_names structure which holds the data copied by getname().
- auditable: This field is set to 1 if the audit_context needs to be written on syscall exit.
- pwd: Current working directory from where the task has started.
- pwdmnt: Current working directory mount point. pwdmnt and pwd are used to set the cwd field of FS_WATCH audit record type.
- aux: A pointer to auxiliary data structure to be used for event specific audit information.
- pid: Process id.
- arch: The machine architecture.
- personality: The OS personality number.
- Other fields: The audit context also holds the various user and group real, effective, user and file system id's: uid, euid, suid, fsuid, gid, egid, sgid, fsgid.

5.6.1.2 File system audit components

File system auditing is implemented using of the inotify kernel file modification notification system (Section 5.1.4). The kernel audit subsystem initialization routine audit_init() registers a vector of inotify operations using the inotify_init() function. The operations vector contains the audit subsystem inotify event notification function audit_handle_ievent() and the audit subsystem inotify destroy function audit_free_parent(). The audit subsystem inotify handle is returned by a successful audit_init() call. When audit inotify events occur, the audit_handle_event() updates audit context inode data to reflect changes in watched file status.

When the audit subsystem receives an instruction from auditctl to set a watch on a file system object, the audit_recieve_skb() function receives the netlink packet in the kernel. It in turn calls audit_receive_message(), which dispatches the appropriate function based upon the operation requested. For audit rule updates, it calls audit_receive_filter(). The audit_receive_filter() routine calls audit_data_to_entry(), which converts the audit data to a watch and calls audit_to_watch() to initialize the audit watch data structure, and then calls audit_add_rule(). The audit add_rule_function() adds the inotify watch for the watch rule by calling audit_add_watch(), which scans the list of active audit inotify watch parents and adds the parent if it does not already exist by calling audit_init_parent(). The audit_init_parent() function calls inotify_init_watch() and inotify_add_watch() to initialize the inotify watch and register it with the inotify subsystem. It finally adds the watch to the parent by calling the audit_add_to_parent() function, which associates the watch rule with the watch parent.

When a filesystem object the audit subsystem is watching changes, the inotify subsystem calls the `audit_handle_event()` function. `audit_handle_event()` in turn updates the audit subsystem's watch data for the watched entity. This process is detailed in Section 5.6.3.1.3.

5.6.1.3 User space audit components

The main user level audit components consist of a daemon (`auditd`), a control program (`auditctl`), a library (`libaudit`), a configuration file (`auditd.conf`), and an initial setup file (`auditd.rules`). There is also an `init` script that is used to start and stop `auditd`, `/etc/init.d/auditd`. When run, this script sources another file, `/etc/sysconfig/auditd`, to set the locale, and to set the variable `AUDIT_CLEAN_STOP`, which controls whether to delete the watch points and the filter rules when `auditd` stops.

On startup, `auditd` reads the configuration file to set the various configuration options that pertain to the daemon. Then, the `auditd` reads the `auditd.rules` to set the initial rules. The `auditd.conf` man page describes all the configurable options, and the `auditctl` man page lists all the supported control options.

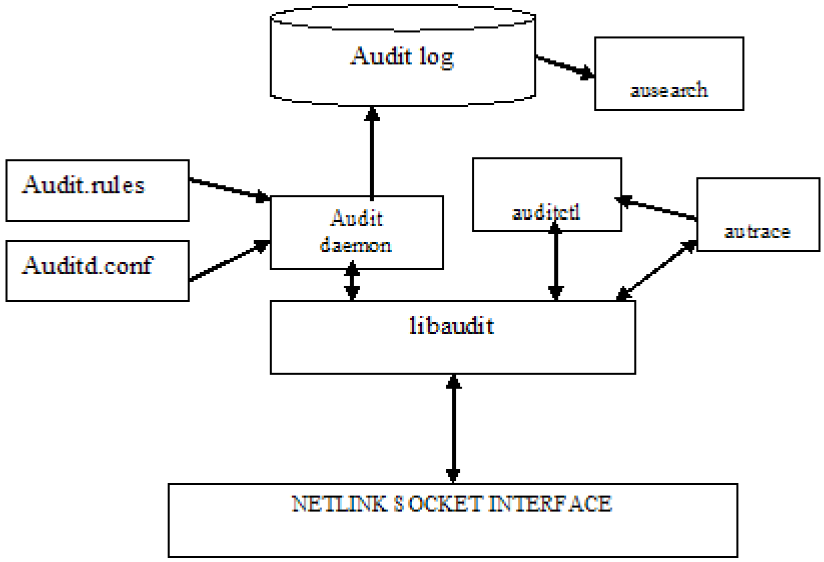


Figure 5-72: Audit User Space Components

5.6.2 Audit operation and configuration options

5.6.2.1 Configuration

There are many ways to control the operation of the audit subsystem. The controls are available at compilation time, boot time, daemon startup time, and while the daemon is running.

At compilation time, SLES kernel provides three kernel configuration options that control the level of audit support compiled into the kernel. The options are:

- `CONFIG_AUDIT`: This enables the base level of audit support.
- `CONFIG_AUDITSYSCALL`: This enables the ptrace hooks for the full syscall audit trace. The currently supported architectures include X86, PPC64, S390x, IA64, X86_64.
- `CONFIG_AUDITFILESYSTEM`: This enables file system auditing.

At boot time, LAF provides the option audit, which enables the system call and file system auditing support. If audit is set to 1, system call and file system auditing are enabled; otherwise, both system call and file system auditing are disabled. After the system is up and running, the administrator has the ability to enable and disable syscall and file system auditing by using `auditctl` with the `-e` option.

On startup, `auditd` reads the `/etc/auditd.conf` file, which holds options that can be set by the administrator to control the behavior of the daemon. Table 5-2 lists the various configuration options. In addition, `auditd` reads `/etc/audit.rules` file, which holds any command supported by `auditctl`. The `auditd` and `auditctl` man pages give more detailed info.

Option	Description	Possible values
log_file	name of the log file	
log_format	How to flush the data from auditd to the log.	RAW. Only RAW is supported in this version.
priority_boost	The nice value for auditd. Used to run auditd at a certain priority.	
flush	Method of writing data to disk.	none, interval, data, sync
freq	Used when flush is incremental, states how many records written before a forced flush to disk.	
num_logs	Number of log files to use	
max_log_file	Maximum log size in megabytes.	
max_log_file_action	Action to take when the maximum log space is reached.	ignore, syslog, suspend, rotate
space_left	Low water mark	
space_left_action	What action to take when low water mark is reached	ignore, syslog, suspend, single, halt
admin_space_left	High water mark	
admin_space_left_action	What action to take when high water mark is reached	ignore, syslog, suspend, single, halt
disk_full_action	What action to take when disk is full	ignore, syslog, suspend, single, halt
disk_error_action	What action to take when an error is encountered while writing to disk.	

Table 5-2: /etc/auditd.conf options

In addition to setting the audit filter rules, auditctl can be used to control the audit subsystem behavior in the kernel even when auditd is running. These settings are listed in Table 5-3.

Option	description	Possible values
`-b`	Sets max number of outstanding buffer allowed. If all buffers are exhausted, the failure flag is checked.	Default is 64
`-e`	Sets enabled flag	0|1
`-f`	Sets failure flag	`silent`, `printk`, `panic`
`-r`	Sets the rate of messages/second. If 0 no rate is set. If > 0 and rate exceeded, the failure flag is checked.	

Table 5-3: `auditctl` control arguments

5.6.2.2 Operation

The audit framework operation consists of the following steps:

On kernel startup, if audit support was compiled into the kernel, the following are initialized:

1. The netlink socket is created.
2. Four lists that hold the filter rules in the kernel are initialized.
3. The `audit_initialized` flag is set to true.
4. The `audit_enabled` flag is set according to `audit_default` or to the boot parameter `audit`. No syscall or file system auditing takes place without `audit_enabled` being set to true. The `audit_enabled` flag can also be set from the command line using `auditctl -e 1`.
5. The file system auditing is initialized by creating the watch lists and the hash table for the file system auditing.

`auditd` does the following on startup:

1. Registers its `pid` with the kernel, so that the kernel starts sending all audit events to the daemon (to the netlink).
2. Enables auditing.
3. Opens the netlink socket, and spawns a thread that continuously waits on the condition of audit record data availability on the netlink. Once the data is available, it signals the thread that writes out the audit records.
4. Reads the `/etc/auditd.conf` configuration file, which holds the configuration parameters that define, among other things, what to do when errors are encountered, or when the log files are full.
5. Usually, `auditd` is run by the `/etc/init.d/auditd` initialization script, which issues `auditctl -R /etc/audit.rules`, if the file `/etc/auditd.rules` exists.
6. The `auditctl` command can be used at any time, even before `auditd` is running, to add and build rules associated with possible actions for system calls and file system operations, as well as setting the behavior of the audit subsystem in the kernel.

7. If audit is enabled, the kernel intercepts the system calls, and generates audit records according to the filter rules. Or, the kernel generates audit records for watches set on particular file system files or directories.
8. Trusted programs can also write audit records for security relevant operation via the audit netlink, not directly to the audit log.
9. The `auditctl -m` option is another way to write messages to the log.

5.6.3 Audit records

The kernel as well as user space generate the audit records. The records can take various formats depending on the information they hold and from where they are generated. This section highlights the areas from which the audit records are generated, and the various audit record formats.

5.6.3.1 Audit record generation

The audit framework is an event-based system in which data is recorded whenever an auditable event occurs. An event represents a single action that might affect the security of the system. Either system calls or user-level trusted programs trigger events.

In order to catch all security-relevant events, audit is designed to record actions of the kernel as well as those of the security-relevant trusted programs. As such, there are two sources that generate audit records: the kernel and trusted user programs.

All actions are recorded in the form of audit records. Each audit record contains information pertaining to the security-relevant action, which allows the administrator to irrefutably attribute the action to a particular user and ascertain the time that the action was taken.

The Audit Subsystem, by default, has the general logging mechanism active. This mechanism offers a set of APIs that can be used by other kernel subsystems, such as SELinux (SELinux is not used in SLES). If the audit daemon is not listening, or Netlink is not configured, the records are logged to syslog via printk.

The following sections describe how records are generated through the kernel and user space.

5.6.3.1.1 Kernel record generation

The kernel component of Linux audit consists of extensions to the process task structure for storing additional audit related attributes. The kernel provides intercept functions, or hooks, to trap entry into and exit from the system calls interface, hooks to the VFS to track file system related events, and hooks for IPC and socket operations. The kernel evaluates the security events and, if allowed by the filters, generates audit records.

To begin auditing, a process attaches itself to the audit framework. Attaching means the process extends the task structure to include an audit context. By default, the task builds the audit context, if audit is enabled. Only when specifically stated that auditing should be disabled for the task, the audit context will not be built for the task. As mentioned in the Linux Audit Operation section, at kernel initialization, four lists of rules are created. The per-task list holds filter rules that are checked on task creation to see whether auditing should be disabled for the process.

An audit context can be in one of the following four states:

- Disabled, no syscall audit will be generated for the task. The audit context is NULL.
- Build the audit context, but do not fill in the sycall info at syscall entry.
- Build the audit context and always fill it on syscall's entry.
- Build the audit context, fill it at syscall's entry, and always write it out at syscall's exit.

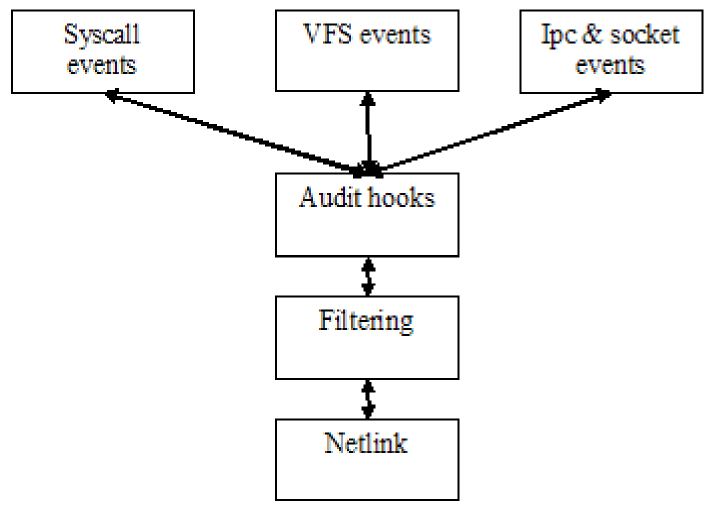


Figure 5-73: Audit Record Generation

5.6.3.1.2 Syscall audit record generation

Once attached, every security-relevant system call performed by the process is evaluated in the kernel. The process's descendants maintain their attachment to the audit subsystem.

1. All security-relevant system calls made by the process are intercepted at the beginning or at the exit of the system call code. The intercept routine then evaluates the intended action, and an audit context corresponding to the system call is built to be used in the corresponding audit record.
2. The process invokes a system call service routine to perform the intended system call.
3. The audit record is placed in a netlink; from there, it is transferred to the audit trail by the audit daemon.

5.6.3.1.2.1 System call interface intercept functions

In order to evaluate each system call as a potential audit candidate, the SLES kernel's system call interface functions are extended with calls to audit framework functions. Ordinarily, system calls are performed in a three step process.

The first step changes from user to kernel mode, copies system call arguments, and sets up appropriate kernel registers. The second step calls the system call service routine to perform the system call, and the third step switches from the kernel to user space after copying the result of the system call.

The LAF extensions to the previous steps involve calling the audit subsystem function `audit_syscall_entry()` between step one and two, and calling the audit subsystem function `audit_syscall_exit()` between steps two and three. This is done by adding hooks to the ptrace system. The `do_syscall trace()` function is called twice, once before the system call is performed, at which time it calls `audit_syscall_entry()` and once again after the system call is performed. At that time it calls `audit_syscall_exit()`. `audit_syscall_entry()` stores relevant audit data needed to create the audit record. `audit_syscall_exit()`, which is invoked after the system call is performed, generates the audit record, and sends the record to netlink socket. Both `audit_syscall_entry()` and `audit_syscall_exit()` call `audit_filter_syscall()` to apply filter logic, to check whether to audit or not to audit the call.

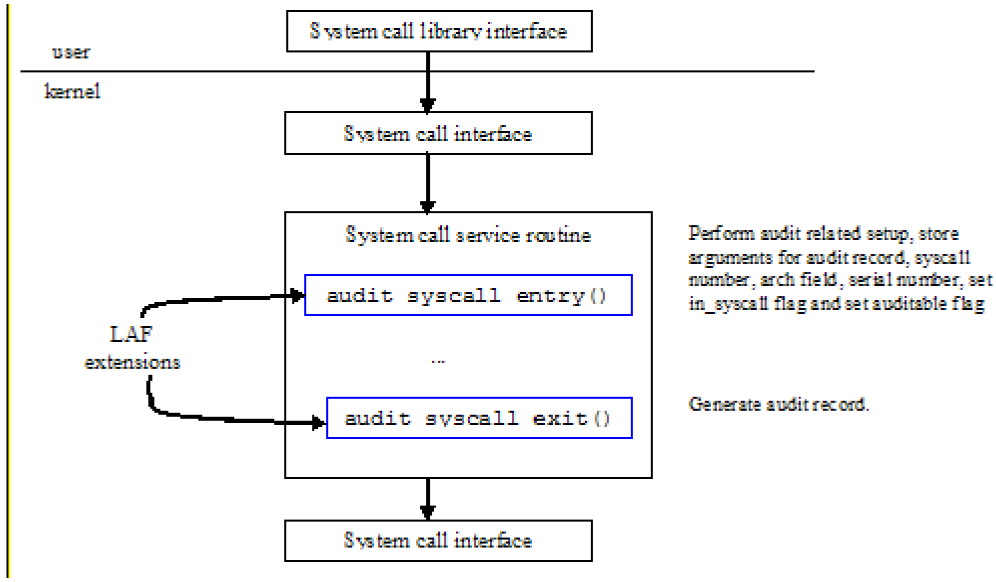


Figure 5-74: Extension to system calls interface

Filtering logic allows an administrative user to filter out events based on the rules set, as described in the `auditctl` man page.

5.6.3.1.3 File system audit record generation

The file system code is hooked with the inotify subsystem. The audit subsystem registers its notification function `audit_handle_ievent()`, which the inotify subsystem calls when a security-relevant operation occurs on a watched object. The notification function covers creation, deletion, moving, renaming, link, and unlink,

Permission changes, as well as access and modification of the object security attributes, `chown`, `chmod`, `setxattr`, and `removexattr` are audited by `audit_inode()` hooks inserted into the system calls. The hooks directly update the inode information in the audit context.

When a watched object is accessed by a system call, the audit subsystem's information about the inode and its watches is updated. A typical sequence of file system operations that generates audit records for a watched object follows these steps:

1. A system call is entered.
2. The system call modifies a watched file's inode information, triggering an inotify event that calls the `audit_handle_ievent()` function with the inotify watch event information, which updates the audit context's inode information; or, in certain cases, a hooked system call updates the audit context's inode information.
3. At syscall exit, `audit_log_exit()` detects the updated inode information in the audit context and emits `PATH` and `SYSCALL` records for the watch event via the audit netlink interface.

5.6.3.1.4 Socket call and IPC audit record generation

Some system calls pass an argument to the kernel specifying which function the system call is requesting from the kernel. These system calls request multiple services from the kernel through a single entry point. For example, the first argument to the `ipc()` call specifies whether the request is for semaphore operation, shared memory operation, and so forth. In the same manner, the `socketcall()` system call is a common kernel entry point for the socket system calls. The `socketcall()` and the `ipc()` call are extended to audit the arguments and therefore audit the exact service being performed. Following is a typical flow:

1. The kernel encounters a `socketcall()` or `ipc()` call.
2. The kernel invokes an audit framework function to collect appropriate data to be used in the auxiliary audit context.
3. The call is processed.
4. On exit the audit record that includes the auxiliary audit information is placed on the netlink.

5.6.3.1.5 Record generation by trusted programs

Trusted programs create their own audit records in which their actions are described. The following describes a typical trusted program operation with respect to audit:

To begin auditing after a security relevant action, a trusted process opens the netlink socket, checks whether the audit is enabled, formats an audit message describing the action, then writes the audit message to the kernel. In turns, the kernel checks the user message filter list and, if appropriate, it sends the audit message back to `auditd` for logging.

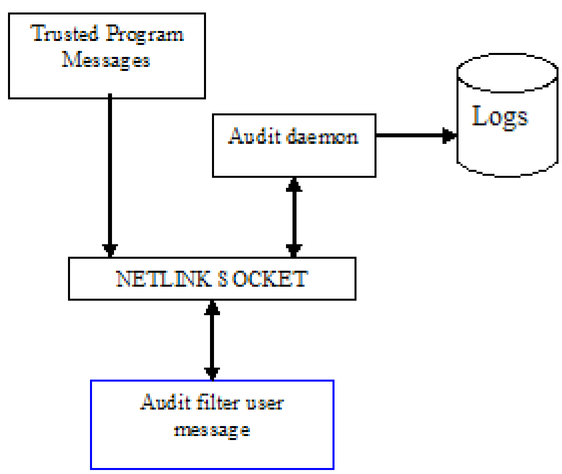


Figure 5-75: User Space Record Generation

5.6.3.2 Audit record format

Each audit record consist of record type, a time stamp, login ID, and process ID, along with variable audit data depending on the audit record type. In other words, the record depends on the audit event. Since audit records are written to user-space as soon as they are generated, a complete audit record might be written in several pieces. A time stamp and a serial number pair identify the various pieces of the audit records. The timestamp of the record and the serial number are used by the user-space daemon to determine which pieces belong to the same audit record. The tuple is unique for each syscall and lasts from syscall entry to syscall exit. The tuple is composed of the timestamp and the serial number.

Each audit record for system calls contain the system call return code, which indicates if the call was successful or not. The following table lists security relevant events for which an audit record is generated on the TOE.

Event Description	LAF audit events
Startup and shutdown of audit functions	DAEMON_START, DAEMON_END are generated by auditd
Modification of audit configuration files	DAEMON_CONFIG, DAEMON_RECONFIG are generated by auditd. Syscalls open, link, unlink, rename, truncate, write on configuration files
Successful and unsuccessful file read/write	Syscall open
Audit storage space exceeds a threshold	space_left_action, admin_space_left_action configuration parameters for auditd.
Audit storage space failure	disk_full_action, disk_error_action configuration parameters for auditd.
Operation on file system objects	Syscalls chmod, chown, setxattr, removexattr, link, symlink, mknod, open, rename, truncate, unlink, rmdir, mount, umount, semtimedop
Operations on message queue	Syscalls msgctl, msgget
Operations on semaphores	Syscalls semget, semctl, semop, semtimedop.
Operations on shared memory segments	Syscalls shmget, shmctl
Rejection or acceptance by the TSF of any tested secret.	Audit record type: USER_AUTH from PAM framework and audit record type: USER_CHAUTHTOK from shadow utilities.
Use of identification and authentication mechanism	Audit record type: USER_AUTH, USER_CHAUTHTOK from PAM framework.
Success and failure of binding user security attributes to a subject (e.g. success and failure to create a subject)	Audit record type: LOGIN from pam_login.so module. Syscalls: fork and clone.
All modification of subject security values	Syscalls chmod, chown, setxattr, msgctl, semctl, shmctl, removexattr, truncate
Modifications of the default setting of permissive of restrictive rules	Syscalls umask, open
Modification of TSF data	Syscalls open, rename, link, unlink, truncate, chmod, chown, setxattr, removexattr (of audit log files and audit configuration files), messages from shadow suites, audit record type: USER_CHAUTHTOK.
Modifications to the group of users that are part of a role	Audit messages from trusted programs in the shadow suite, audit record type: USER_CHAUTHTOK.

Event Description	LAF audit events
Execution of the test of the underlying machine and the result of the test	Audit message from amtu utility: audit record type: USER.
Changes to system time	Syscall settimeofday, adjtimex
Setting up a trusted channel	Sycall exec (of stunnel program)

Table 5-4: Audit Subsystem event codes

5.6.4 Audit tools

In addition to the main components, the user level provides a search utility, ausearch, and a trace utility, autrace. While ausearch finds audit records based on different criteria from the audit log, autrace audit all syscalls issued by the process being traced. The man pages for these two utilities detail all the options that can be used for each. In this section we briefly describe how they operate.

5.6.4.1 auditctl

The auditctl command configures and examines the kernel audit subsystem. It allows the setting of syscall rules, file watches, various audit characteristics, and the sending of userspace messages. It communicates with the kernel using the netlink socket interface via the audit library. For more information on auditctl, please see the auditctl(8) man page. Use of auditctl is restricted in the TOE to administrative users.

5.6.4.2 ausearch

Only root has the ability to run this tool. First ausearch checks the validity of the parameters passed, whether they are supported or not. Then it opens either the logs or the administrator-specified files. The logs' location is extracted from the /etc/auditd.conf. After that, ausearch starts to process the records, one record at a time, matching the parameters passed to it. Each audit record can be written into the log as multiple file records. The tool collates all the file records into a linked list before it checks whether the record matches the requested search criteria. For more information on ausearch, please see the ausearch(8) man page.

5.6.5 Login uid association

The pam_loginuid.so module writes the login uid of the process that was authenticated to the /proc system (/proc/session id/loginuid). The loginuid file is only writable by root and readable by everyone. The /proc file system triggers the kernel function audit_set_loginuid() to set the login uid for the user in the audit context. From then on, this login uid is maintained throughout the session to trace back all operations done in the session to exactly the login user.

5.7 Kernel modules

Kernel modules are pieces of object code that can be linked to, and unlinked from, the kernel at runtime. Kernel modules usually consist of a set of functions that implement a file system, a device driver, or other functions at the kernel's upper layer.

Lower-layer functions, such as scheduling and interrupt management, cannot be modularized. Kernel modules can be used to add or replace system calls. The SLES kernel supports dynamically-loadable kernel modules that are loaded automatically on demand. Loading and unloading occurs as follows:

1. The kernel notices that a requested feature is not resident in the kernel.
2. The kernel executes the `modprobe` program to load a module that fits this symbolic description.
3. `modprobe` looks into its internal alias translation table to see if there is a match. This translation table is configured by alias lines in `/etc/modprobe.conf`.
4. `modprobe` then inserts the modules that the kernel needs. The modules are configured according to options specified in `/etc/modprobef.conf`.

By default, the SLES system does not automatically remove kernel modules that have not been used for a period of time. The SLES system does, however, provide a mechanism by which an administrator can periodically unload unused modules.

Each module automatically linked into the kernel has the `MOD_AUTOCLEAN` flag set in the flags field of the module object set. The administrator can set up a cron job to periodically execute `rmmod -a` to tag unused modules as to be cleaned, and to remove already tagged modules. Modules stay tagged if they remain unused since the previous invocation of `rmmod -a`. This two-step cleanup approach avoids transiently unused modules.

Only root administrator can modify the `/etc/modprobe.conf` file, allowing the administrator complete control over which modules can be loaded, and with what configuration options. In the kernel, the `sys_create_module()` (for manual load operation) and `sys_init_module()` (called by `modprobe` during automatic load operation) module load functions are protected by a process uid check of 0. Thus, only a privileged process can initiate the loading of modules in the kernel.

5.7.1 Linux Security Module framework

The Linux kernel (from 2.6 onwards) provides a general kernel framework to support security modules. In particular, the Linux Security Module (LSM) framework provides the infrastructure to support access control modules.

The LSM kernel patch adds security fields to kernel data structures, and inserts calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. It also adds functions for registering and unregistering security modules, and adds a general security system call to support new system calls for security-aware applications. The LSM `security` fields are simply void* pointers. Table 5-5 shows the kernel data structures that the LSM kernel patch modifies, and the corresponding abstract objects.

STRUCTURE	OBJECT
`task_struct`	Task(Process)
`linux_binprm`	Program
`super_block`	File system
`inode`	Pipe, File, or Socket
`file`	Open File
`sk_buff`	Network Buffer(Packet)
`net_device`	Network Device
`kern_ipc_perm`	Semaphore, Shared Memory Segment, or Message Queue
`msg_msg`	Individual Message

Table 5-5: Kernel data structures modified by the LSM kernel patch and the corresponding abstract objects

The `security_operations` structure is the main security structure that provides security hooks for various operations such as program execution, file system, inode, file, task, netlink, UNIX domain networking, socket, and System V IPC mechanisms.

Each LSM hook is a function pointer in a global table, `security_ops`. This table is a `security_operations` structure defined in `include/linux/security.h`. The global `security_ops` table is initialized to a set of hook functions provided by a dummy security module that provides traditional super user logic. A register_security function (in `security/security.c`) is provided to allow a security module to set `security_ops` to refer to its own hook functions, and an `unregister_security()` function is provided to revert `security_ops` to the dummy module hooks. This mechanism is used to set the primary security module, which is responsible for making the final decision for each hook.

LSM also provides a simple mechanism for stacking additional security modules with the primary security module. It defines `register_security()` and `unregister_security()` hooks in the `security_operations` structure, and provides `mod_reg_security()` and `mod_unreg_security()` functions that invoke these hooks after performing some sanity checking. A security module can call these functions in order to stack with other modules.

LSM hooks fall into two major categories: hooks that are used to manage the security fields and hooks that are used to perform access control. Examples of the first category of hooks include the `alloc_security()` and `free_security()` hooks defined for each kernel data structure that has a security field. These hooks are used to allocate and free security structures for kernel objects. The first category of hooks also includes hooks that set information in the security field after allocation, such as the `post_lookup()` hook in `struct inode_security_ops`. This hook is used to set security information for inodes after successful lookup operations. An example of the second category of hooks is the permission hook in `struct inode_security_ops`. This hook checks permission when accessing an inode.

Figure 5-76 depicts the LSM hook architecture in which an LSM hook makes a call to a function that the LSM module (xyz) must provide, just before the kernel would have accessed an internal object. The module can either let the access occur, or deny access, forcing an error code return.

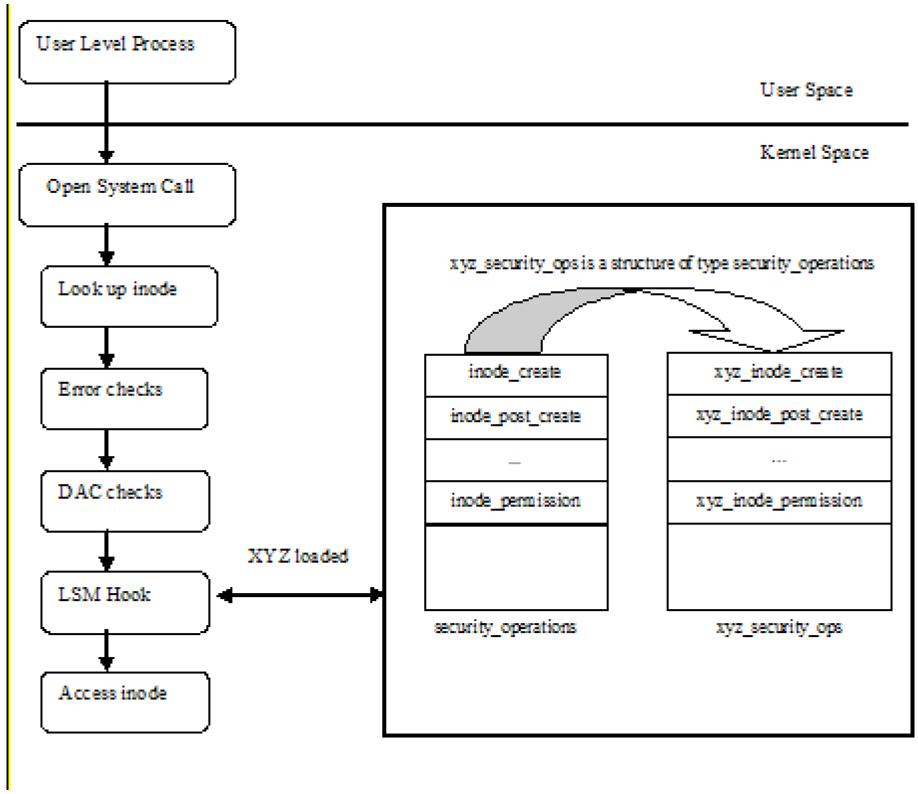


Figure 5-76: LSM hook architecture

LSM adds a general security system call that simply invokes the `sys_security` hook. This system call and hook permits security modules to implement new system calls for security-aware applications.

5.7.2 LSM capabilities module

The LSM kernel patch moves most of the existing POSIX.1e capabilities logic into an optional security module stored in the file `security/capability.c`. This change allows users who do not want to use capabilities to omit this code entirely from their kernel, instead using the dummy module for traditional super user logic or any other module that they desire.

The capabilities logic for computing process capabilities on execve and set*uid, checking capabilities for a particular process, saving and checking capabilities for netlink messages, and handling the `capget` and `capset` system calls, have been moved into the capabilities module.

5.7.3 LSM AppArmor module

SLES includes AppArmor, which is an add-on security module. AppArmor provides application containment functionality. AppArmor is described in the next section. For more information about AppArmor, see http://www.novell.com/linux/security/apparmor/ .

5.8 AppArmor

The AppArmor logical subsystem consists of

- The AppArmor access control functions, provided by the AppArmor LSM module. The AppArmor LSM modules provides security functions that implement additional access control checks when subjects access objects.

- Administrative utilities provide a mechanism for administrators to configure, query, and control AppArmor.

For background information on AppArmor which was originally named SubDomain, *SubDomain: Parsimonious Server Security* by Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor at https://forgesvn1.novell.com/viewsvn/apparmor/trunk/docs/papers/subdomain-lisa00.pdf?revision=3 [CRISP] and http://www.novell.com/documentation/apparmor/pdfdoc/apparmor2_admin/apparmor2_admin.pdf and http://forge.novell.com/modules/xfmod/project/?apparmor .

5.8.1 AppArmor administrative utilities

The primary configuration file for AppArmor is `/etc/apparmor/subdomain.conf` . (SubDomain was the original name for AppArmor.) The configuration file defines the directory where AppArmor profiles are located, what action to take if the AppArmor LSM cannot be loaded at system boot time (`warn`, `panic`, `build`, or `build-panic`), whether the OWLSM extension should be loaded, and whether event logging should occur. For more information about AppArmor configuration, please see the man page on subdomain.conf.

AppArmor profiles define the confinement rules for applications protected by AppArmor. The profiles are kept in `/etc/apparmor.d`.. Profiles are named by the full path to the executable with / replaced by a period (.). The following contains an example AppArmor policy for `klogd` which is stored in `/etc/apparmor.d/sbin.klogd`:

```
#include <tunables/global>

/sbin/klogd {

        #include <abstractions/base>

        capability sys_admin,

        /boot/System.map*        r,

        /proc/kmsg               r,

        /sbin/klogd              rmix,

        /var/log/boot.msg        rwl,

        /var/run/klogd.pid       rwl,

}
```

In this example, klogd, can read the specified files in `/boot/System.map*` and `/proc/kmsg. klogd` can write log and run information, such as `/var/log/boot.msg` and `/var/run/klogd.pid.` Allowable access is denoted by familiar UNIX permission contructs, with some additions, as follows:

- r - read
- w - write
- ux - unconstrained execute
- Ux - unconstrained execute after scrubbing the environment

- px - discrete profile execute
- Px - discrete profile execute after scrubbing the environment
- ix - inherit execute
- m - allow PROT_EXEC with mmap(2) calls
- l – link

For more information about complete AppArmor profile syntax, please see the apparmor.d man page.

AppArmor profiles are loaded into the kernel by the `apparmor_parser` tool. `apparmor_parser` can load new profiles, replace profiles, and remove profiles. Profiles can optionally and individually be selected to be loaded in "`Complain`" mode so that AppArmor does not enforce the profile but just logs an error message if access would be denied by AppArmor with the profile. For more information on `apparmor_parser`, see the `apparmor_parser` man page.

AppArmor also provides a status tool, `apparmor_status`. `apparmor_status` provides information about the number of profiles loaded in enforcing and complaining mode and the number of running processes being confined by AppArmor. For more information on `apparmor_status` please see the `apparmor_status` man page.

The `confined` program reports which programs with open network sockets are running without the protection of an AppArmor profile. The `complain` program allows an authorized administrator to switch AppArmor out of enforcing mode and into complaining mode for a targeted program. The `enforce` program allows an authorized administrator to do the opposite, switch from complain to enforcing mode for a particular profile. `genprof` can be used to generate a profile with all of the permission that were exercised during a test run of the targeted program. Please see the confined, enforce, complain, and genprof man pages for more detail.

For an application contained by an AppArmor profile, access that is not explicitly allowed is denied.

5.8.2 AppArmor access control functions

AppArmor access control functions are called through LSM hooks from various points in the kernel when new subjects and objects are created, when access between subject and object is mediated, and when subject and object security attributes transition to different values (such as during an `execve()` call). The AppArmor profile is applied to a process during the `execve()` call. If an AppArmor profile for an executable is loaded after instances of that executable have already started running, the preexisting processes will not be confined by AppArmor. Please see the `apparmor` man page for additional detail.

5.8.3 securityfs

Communication between the AppArmor kernel component and the AppArmor administrative utilities takes place through the securityfs interface, mounted at `/sys/kernel/security/apparmor`. `apparmor_parser` uses `/sys/kernel/security/apparmor/.load` to load new profiles and likewise uses `/sys/kernel/security/apparmor/.replace` and `/sys/kernel/security/apparmor/.remove` to replace and remove profiles. `apparmor_status` uses `/sys/kernel/security/apparmor/profiles` to generate the status report.

5.9 Device drivers

A device driver is a software layer that makes a hardware device respond to a well-defined programming interface. The kernel interacts with the device only through these well-defined interfaces. For detailed information about device drivers, see *Linux Device Drivers, 2nd Edition*, by Alessandro Rubini and Jonathan Corbet.

The TOE supports many different I/O devices, such as disk drives, tape drives, and network adapters. Each of these hardware devices can have its own methods of handling data.

The device driver subsystem provides a layer of abstraction to other kernel subsystems, so they can interact with hardware devices without being cognizant of their internal workings. Each supported hardware device has a device driver that is loaded into the kernel during system initialization. The device driver subsystem provides access to these supported hardware devices from user space through special device files in the `/dev` directory. Valid operations on the device-special files are initialized to point to appropriate functions in the device driver for the corresponding hardware device.

Other kernel subsystems, such as File, I/O, and the Networking subsystems, have direct access to these device driver functions because the device driver is loaded into the kernel space at system initialization time. The File, I/O, and Networking subsystems interact with these device driver functions to drive the hardware device.

For example, when a file is to be written to a hard drive, data blocks corresponding to the file are queued in the buffer cache of the File and I/O subsystem. From there, the File and I/O subsystem invokes the function to flush the data to the desired device. The device driver corresponding to that device then takes that data and invokes the device-specific functions to transfer the data to the hard drive. Similarly, the Networking subsystem interacts with the device driver subsystem to transfer networking traffic to a network adapter. The physical layer of the networking stack invokes appropriate functions to send and receive networking packets through a network adapter. The device driver corresponding to the network adapter invokes appropriate adapter-specific functions to send or receive network packets through the network adapter.

Device drivers provide a generic interface to the rest of the kernel consisting of device methods for the start-up of a device (open method), shutdown of a device (release method), flushing contents of internal buffers (flush method), reading data from the device (read method), writing data to the device (write method), and performing device-specific control operations (ioctl method).

SLES running on System p and System z supports virtual devices. From the perspective of SLES, these devices are treated no differently than other devices. That is, the SLES kernel thinks that it is directly controlling devices. However, the z/VM on System z map these virtual devices to real devices, allowing SLES access to devices supported by z/VM when running on System z. The following subsections briefly describe this virtualization of I/O, followed by brief descriptions of device drivers for character and block devices.

5.9.1 I/O virtualization on System z

SLES runs on System z as a guest of the z/VM operating system. The z/VM operating system can provide each end user with an individual working environment known as virtual machine. Virtual machines communicate with the outside world through virtual devices. The Control Program transparently handles mapping of virtual to real devices and resources.

5.9.1.1 Interpretive-execution facility

The interpretive-execution facility provides complete handling of many aspects of the architecture of an interpreted machine or, when such handling is not provided, presents virtual-machine status in a form convenient for further support-program processing.

The interpreted machine is viewed as a virtual system called the guest. The term host refers to the real machine and to the control program, which both manages real-machine resources, and provides services to the guest program or interpreted machine. The interpreted and host machines execute guest and host programs, respectively.

The interpretive-execution facility is invoked by executing the Start Interpretive Execution (SIE) processor instruction, which causes the CPU to enter the interpretive-execution mode and to begin execution of the guest program under control of the operant of the instruction, called the state description.

Certain operations encountered in the guest cannot be performed in the interpretive-execution mode, and some may optionally have been designated to cause interpretive-execution to be discontinued. In these cases, the CPU exits from the interpretive-execution mode, the execution of the SIE instruction is completed, and the instruction in the host program that follows the SIE instruction, or that follows an Execute instruction, as appropriate, is designated as the next instruction to be executed. This process is called interception, and it includes saving the state of the guest in the state description and providing information about the reason for exiting from the interpretive-execution mode.

The CPU may also exit from the interpretive-execution mode by interrupting (a host I/O interruption, for example) execution of the SIE instruction. The SIE instruction is an interruptible instruction; that is, there are parameters in the state description for continuing in such a way that if the host old PSW is loaded, causing the SIE instruction (or an Execute instruction as appropriate) to be executed again, execution resumes at the interrupted point in the guest.

The state description specifies:

- the type of system to be interpreted
- the area of host storage representing guest main storage
- the contents of some of the program-addressable guest registers
- the addresses of related control tables
- bits for controlling the operation of optional facilities
- areas for displaying information concerning and interception
- information about other aspects of the operation (including the expiry timer that causes the exiting of the interpretive-execution facility by interrupting the SIE instruction upon expiry of the timer)

5.9.1.2 State description

The SIE instruction designates an operant called the state description, which is located in the host real storage. Fields in the state description provide information about the guest initial state on entry to the interpretive-execution mode, provide guest control and status information used in the interpretive-execution mode, and are used to store information about the guest state on exit from the interpretive-execution mode.

After the CPU enters the interpretive-execution mode, it is undefined whether changes to the state description by the channel subsystem or by another CPU will affect the guest. Fetch references to the state description by the channel subsystem or by another CPU may or may not obtain the current state of the guest. However, bits in the field labeled intervention requests and TCH control may be set to ones by means of an interlocked update by on CPU, while the fields are concurrently being used to control interpretive-execution in another CPU, with the assurance that the change will be observed by the CPU in the interpretive-execution mode. The contents of these two fields are said to be dynamically observed or recognized.

The state description contains the following parameters along with others:

- Definition of the guest memory structure in 64 KB increments.
- General registers 14 and 15, and the PSW.
- Residue counter holding the remainder of the time spent in interpretive-execution mode that has not yet been accounted for. This counter is ignored when the interval timer is not active.

- Conditional interceptions refer to functions that are executed for the guest unless a specified condition is encountered that causes control to be returned to the host by the process that called the interception. Following are some of the controls that can cause interception when the related function is handled for the guest:
  - Supervisor Call (SVC) instruction: It is to be specified whether all guest SVC instructions cause an instruction interception or only those for which the effective I field matches a code.
  - Load Control (LCTL) instruction: The state description bits for this instruction are numbered to correspond to control-register numbers. When a bit is one and a guest LCTC instruction designates the corresponding control register in the range of registers to be loaded, instructions-execution is suppressed, and instruction interception is recognized.
  - Interception Controls: A bit value of one results in interception when the associated function is encountered in the guest. The interception code field stores the code that indicates the reason for the interception. The contents of this field are changed only at interception.

5.9.1.3 Hardware virtualization and simulation

The Control Program (CP) maintains all virtual machines by using the SIE instruction supported by the underlying abstract machine.

Access to resources that must be shared among virtual machines is virtualized by the CP. The CP must ensure the following:

- Each accessing virtual machine is provided with a consistent view to the virtualized resource, such as the timer.
- Each accessing virtual machine must be limited to their configured resources. Access to other resources must be prohibited (such as I/O).

Each access to such a virtualized resource causes the SIE instruction to terminate on the requesting processor and to return control back to the CP. Based on CP's internal state information about the virtual machine, the access request is mediated by CP. Upon completion of the request, the return information is forwarded to the requesting virtual machine by the CP according to the rules applicable for z/Architecture systems.

5.9.2 Character device driver

A character device driver is one that transfers data directly to and from a user process without the sophisticated buffering strategies and disk caches. The `file_operations` structure is defined in `linux/fs.h`, and holds pointers to functions defined by the driver that performs various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

Programs operate on character devices by opening their file system entries. Each file system entry contains a major and a minor number, by which the kernel identifies the device. The Virtual File System sets up the file object for the device, and points the file operations vector to `def_chr_fops` table. `def_chr_fops` contains only one method, `chrdev_open()`, which rewrites the `f_op` field of the `file` object with the address stored in the `chrdevs` table element that corresponds to the major number, and minor number if the major is shared among multiple device drivers, of the character device file.

Ultimately the file object's `f_op` field points to the appropriate `file_operations` defined for that particular character device. The `file_operations` structure provides pointers to generic operations that can be performed on a file. Each device driver defines the file operations that are valid for the type of device that the driver manages.

This extra level of indirection is needed for character devices, but not for block devices, because of the large variety of character devices and the operations they support. The following diagram illustrates how the kernel maps the file operations vector of the device file object to the correct set of operations routines for that device.

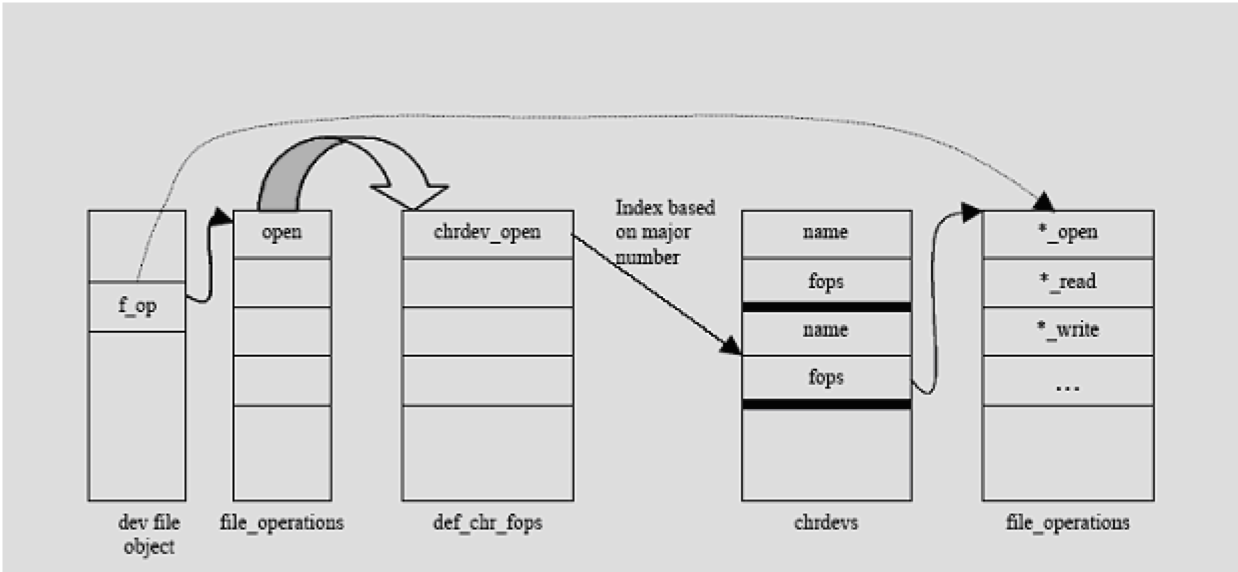


Figure 5-77: Setup of `f_op` *for character device specific file operations*

5.9.3 Block device driver

Block drivers provide access to the block-oriented devices that transfer data in randomly accessible, fixed-size blocks, such as a disk drive. All I/O-to-block devices are normally buffered by the system (the only exception is with raw devices); user processes do not perform direct I/O to these devices.

Programs operate on block devices by opening their file system entries. The file system entry contains a major and a minor number by which the kernel identifies the device. The kernel maintains a hash table, indexed by major and minor number, of block-device descriptors. One of the fields of the block device descriptor is `bd_op`, which is pointer to the block_device_operations structure. The `block_device_operations` structure contains methods to `open`, `release`, `llseek`, `read`, `write`, `mmap`, `fsync`, and `ioctl` that control the block device. Each block device driver needs to implement these block device operations for the device being driven.

The Virtual File System sets up the file object for the device and points the file operations vector to the appropriate block device operations as follows.

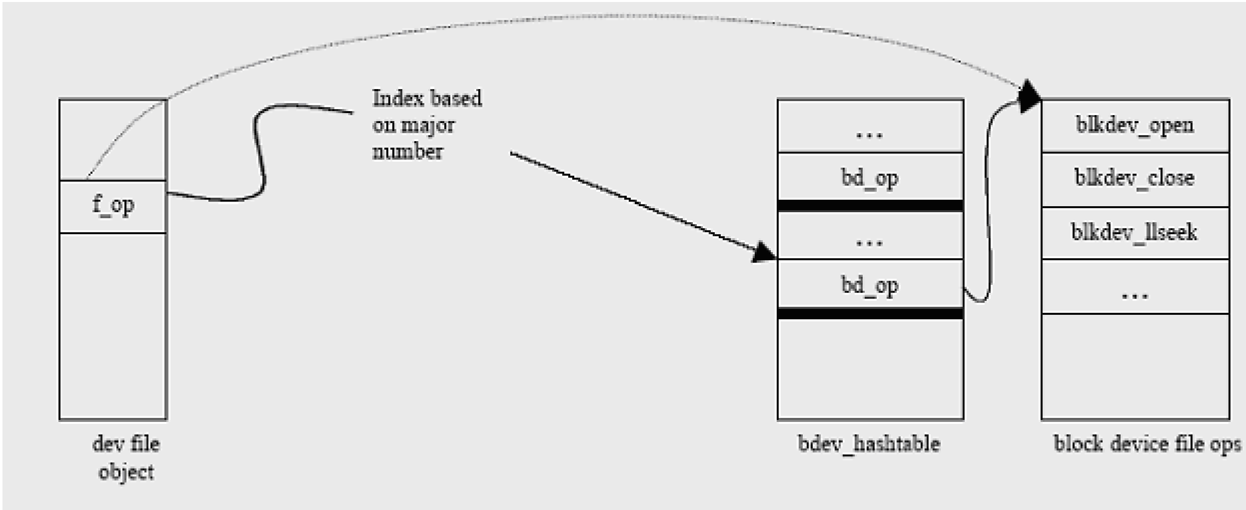


Figure 5-78: Setup of `f_op` for block device specific file operations

5.10 System initialization

When a computer with SLES is turned on, the operating system is loaded into memory by a special program called a boot loader. A boot loader usually exists on the system's primary hard drive, or other media device, and has the sole responsibility of loading the Linux kernel with its required files or, in some cases, other operating systems, into memory.

Each architecture capable of running SLES uses a different boot loader. The following table lists the boot loaders available for architectures relevant to the target of evaluation:

Architecture	**Boot Loaders**
AMD® AMD64	GRUB
IBM eServer System p	YABOOT
IBM eServer System z	z/IPL
x86	GRUB or LILO

Table 5-6: Boot Loaders by Architecture

This section describes the system initialization process of eServer systems. Because part of the initialization is dependent on the hardware architecture, the following subsections identify and describe, where appropriate, how the hardware-dependent part of the system initialization is implemented for System x, System p, System z, and eServer 326 lines of servers, which are all part of the TOE.

5.10.1 init

The `init` process is the ancestor of all userspace processes and is process ID 1. Its main functions are to start child processes based on the contents of `/etc/inittab`, to reap child processes, to switch between the runlevels defined in `/etc/inittab`, and to react to certain signals, such as power failure signals. If it is executed as a PID other than PID 1, it acts as a control program that allows administrative users to manage the system runlevel by controlling PID 1. For more information on the /etc/inittab file, please see the inittab(5) man page. For more information on the init program, please see the init(8) manpage.

The init program generally follows these startup steps:

1. Gets its own name.
2. Sets its umask.
3. Checks for root identity.
4. Checks to see if it is PID 1 (init—the daemon) or not PID 1 (telinit—the control program).
5. If it is telinit, runs telinit processing and exits; continues if it is init.
6. Handles re-execs by setting proc title appropriately, etc.
7. Processes command line arguments.
8.
9. Sets the proc title to the "init boot" message.
10. Begins running init_main() and never exits. Continuing with init_main():
11. Tells the kernel to send Ctrl-Alt-Delete to init for processing.
12. Sets up signal handling.
13. Initializes the console.
14. Sets a default PATH environment variable.
15. Initializes the /var/run/utmp file.
16. Outputs a boot message.
17. Checks for an emergency shell request and if so opens the shell.
18. Reads inittab.
19. Starts child processes.
20. Performs boot transitions.
21. Checks for child processes being waited on.
22. Checks the init fifo.
23. Checks failing flags.
24. Processes signals.
25. Checks once again to see of any children need to be started.
26. Goes back to step 21.

5.10.2 System x

This section briefly describes the system initialization process for System x servers. For detailed information about system initialization, see "Booting Linux: History and the Future" 2000 Ottawa Linux Symposium by Almesberger, Werner and Documentation/i386/boot.txt on a SLES system.

5.10.2.1 Boot methods

SLES supports booting from a hard disk, a CD-ROM, or a floppy disk. CD-ROM and floppy disk boots are used for installation, and to perform diagnostics and maintenance. A typical boot is from a boot image on the local hard disk.

5.10.2.2 Boot loader

A boot loader is a program that resides in the starting sectors of a disk, that is, the Master Boot Record (MBR) of the hard disk. After testing the system during boot, the Basic Input-Output System (BIOS) transfers control to the MBR if the system is set to be booted from there. Then the program residing in MBR gets executed. This program is called the boot loader. Its duty is to transfer control to the operating system, which will then proceed with the boot process.

SLES supports the GRUB (GRand Unified Boot Loader) boot loader. GRUB lets you set pointers in the boot sector to the kernel image and to the RAM file system image. GRUB is a part of the TSF. For detailed information about GRUB, refer to their Web site at http://www.gnu.org/software/grub/ and, on SLES systems, `/usr/share/info/grub.info.gz`.

5.10.2.3 Boot process

The boot process consists of the following steps when the CPU is powered on or reset:

1. The BIOS probes the hardware, establishes which devices are present, and runs the Power-On Self Test (POST). BIOS is not part of the TOE.
2. The BIOS initializes hardware devices and makes sure they operate without IRQ or I/O port conflicts.
3. The BIOS searches for the operating system to boot in an order predefined by the BIOS setting. Once a valid device is found, the BIOS copies the contents of its first sector containing the boot loader into RAM, and starts executing the code just copied.
4. The boot loader is invoked by BIOS to load the kernel and the initial RAM file system into the system's Random Access Memory (RAM). It then jumps to the `setup()` code.
5. The `setup()` function reinitializes the hardware devices in the computer and sets up the environment for the execution of the kernel program. The `setup()` function initializes and configures hardware devices, such as the keyboard, video card, disk controller, and floating point unit.
6. The boot loader reprograms the Programmable Interrupt Controller and maps the 16 hardware interrupts to the range of vectors from 32 to 47. The boot loader switches the CPU from Real Mode to Protected Mode and then jumps to the `startup_32()` function.
7. The boot loader initializes the segmentation registers and provisional stack. It fills the area of uninitialized data of the kernel with zeros.
8. The boot loader decompresses the kernel, moves it into its final position at 0x00100000, and jumps to that address.
9. The boot loader calls the `second startup_32()` function to set the execution environment for process 0.
10. The boot loader initializes the segmentation registers.
11. The boot loader sets the kernel mode stack for process 0.
12. The boot loader initializes the provisional Page Tables and enables paging.
13. The boot loader fills the `bss` segment of the kernel with zeros.

14. The boot loader sets the IDT with null interrupt handlers. It puts the system parameters obtained from the BIOS and the parameters passed to the operating system into the first page frame.
15. The boot loader identifies the model of the processor. It loads the `gdtr` and `idtr` registers with the addresses of the Global Descriptor Table and Interrupt Descriptor Table, and jumps to the `start_kernel()` function.
16. `start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures, IDT tables, the slab allocator (described in Section 5.5.3.6), system date, and system time.
17. The boot loader uncompress the initial RAM file system `initrd`, mounts it, and then executes `/linuxrc`.
18. The boot loader unmounts `initrd`, mounts the root file system, and executes `/sbin/init`. It resets the pid table to assign process ID one to the `init` process.
19. `/sbin/init` determines the default run level from `/etc/inittab` and performs the following basic system initialization by executing the script `/etc/init.d/boot`.
    1. Allows an administrator to perform interactive debugging of the startup process by executing the `/etc/sysconfig/boot` script.
    2. Mounts the `/proc` special file system.
    3. Mounts the `/dev/pts` special file system.
    4. Executes `/etc/init.d/boot.local`, which was set by an administrator to perform site-specific setup functions.
    5. If file /var/lib/YaST2/runme_at_boot exists, finishes YaST2 installation from previous session.
    6. If file /var/lib/YaST2/run_suseconfig exists, executes /sbin/SuSEconfig to configure the operating system.
    7. Performs run-level specific initialization by executing startup scripts defined in `/etc/inittab`. The scripts are named `/etc/init.d/rcX.d`, where X is the default run level. The default run level for a SLES system in the evaluated configuration is 3. The following lists some of the initializations performed at run level 3.
        - Saves and restores the system entropy tool for higher quality random number generation.
        - Configures network interfaces.
        - Starts the system logging daemons.
        - Starts the `sshd` daemon.
        - Starts the `atd` daemon
        - Starts the `cron` daemon.
        - Probes hardware for setup and configuration.
        - Starts the program `agetty`.

        For more detail about services started at run level 3, refer to the scripts in `/etc/init.d/rc3.d` on a SLES system.

For detailed information about the boot process, see http://tldp.org/HOWTO/Linux-Init-HOWTO.html#toc1.

Figure 5-79 schematically describes the boot process of System x servers.

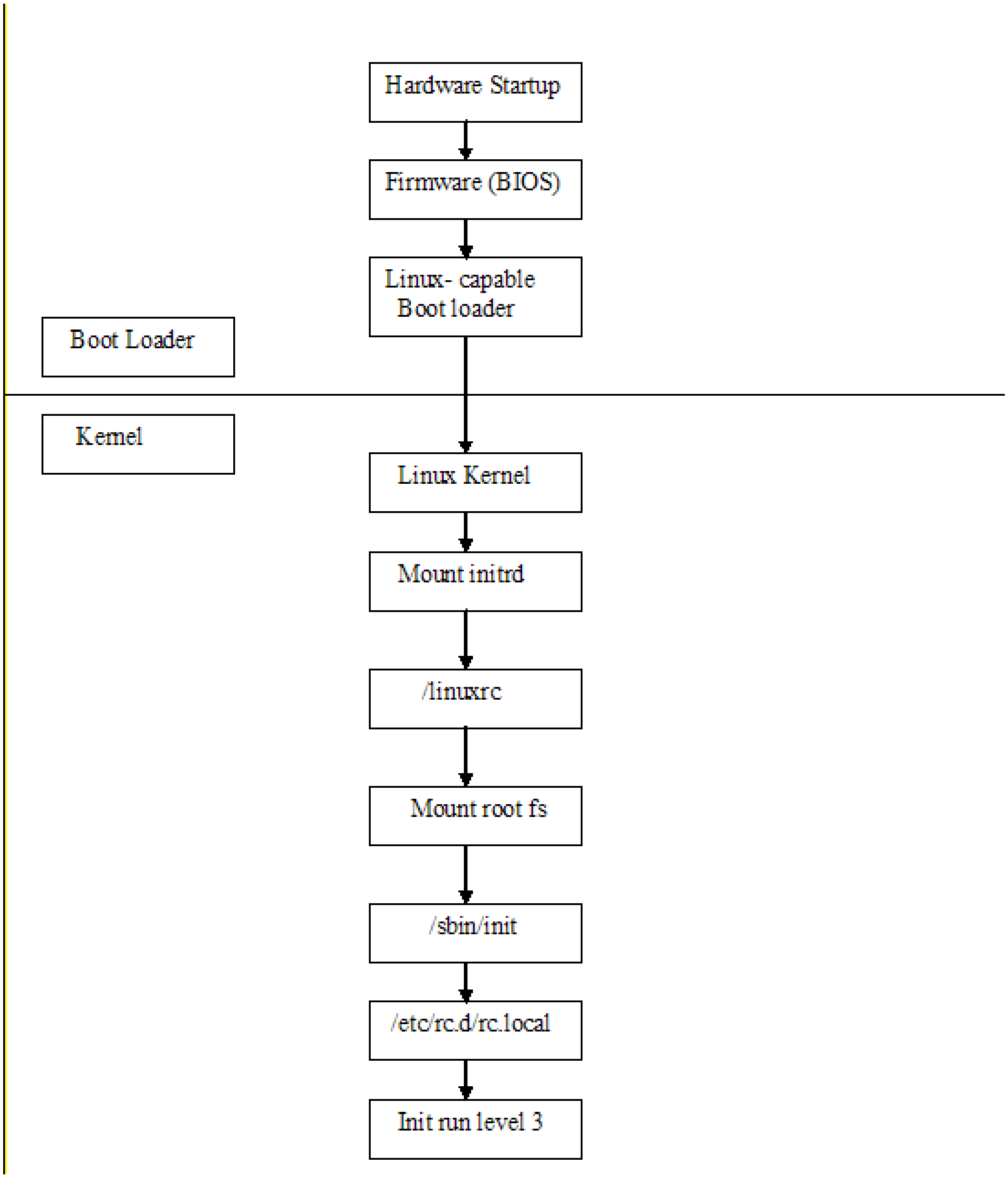


Figure 5-79: System x SLES boot sequence

5.10.3 System p

This section briefly describes the system initialization process for System p servers.

5.10.3.1 Boot methods

SLES supports booting from a hard disk or from a CD-ROM. CD-ROM boots are used for installation and to perform diagnostics and maintenance. A typical boot is from a boot image on the local hard disk.

5.10.3.2 Boot loader

A boot loader is the first program that is run after the system completes the hardware diagnostics setup in the firmware. The boot loader is responsible for copying the boot image from hard disk and then transferring control to it. System p systems boot using a boot loader called Yaboot, which is an abbreviation of Yet Another Boot Loader. Yaboot is an OpenFirmware boot loader for open firmware-based machines. Yaboot is considered to be a part of the TSF. For detailed information on Yaboot, see http://penguinppc.org/bootloaders/yaboot/.

5.10.3.3 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. The system runs power on self tests.
2. Yaboot loads the kernel into a contiguous block of real memory and gives control to it with relocation disabled.
3. Yaboot interacts with OpenFirmware and determines the system configuration, including real memory layout and the device tree.
4. Yaboot instantiates the Run-Time Abstraction Services (RTAS), a firmware interface that allows the operating system to interact with the hardware platform without learning details of the hardware architecture.
5. Yaboot relocates the kernel to real address 0x0.
6. Yaboot creates the initial kernel stack and initializes TOC and NACA pointers.
7. Yaboot builds the hardware page table (HPT) and the segment page table (STAB) to map real memory from 0x0 to HPT itself.
8. Yaboot enables relocation.
9. Yaboot starts kernel initialization by invoking `start_kernel()`.
10. `start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures, IDT tables, the slab allocator (described in Section 5.5.3.6), system date, and system time.
11. Yaboot uncompresses the initial RAM file system `initrd`, mounts it, and then executes `/linuxrc`.
12. Yaboot unmounts `initrd`, mounts the root file system, and executes `/sbin/init`. Yaboot resets the pid table to assign process ID one to the `init` process.
13. `/sbin/init` determines the default run level from `/etc/inittab` and performs the following basic system initialization by executing the script `/etc/rc.d/rc.sysinit`.

1. Yaboot allows an administrator to perform interactive debugging of the startup process by executing the `/etc/sysconfig/init` script.
2. Mounts the `/proc` special file system.
3. Mounts the `/dev/pts` special file system.
4. Executes `/etc/rc.d/rc.local`, which was set by an administrator to perform site-specific setup functions. Performs run-level specific initialization by executing startup scripts defined in `/etc/inittab`. The scripts are named `/etc/rc.d/rcX.d`, where X is the default run level. The default run level for a SLES system in the evaluated configuration is 3. The following lists some of the initializations performed at run level 3.
   - Saves and restores the system entropy tool for higher quality random number generation.
   - Configures network interfaces.
   - Starts the system logging daemons.
   - Starts the `sshd` daemon.
   - Starts the `cron` daemon.
   - Probes hardware for setup and configuration.
   - Starts the `agetty` program.

   For more details about services started at run level 3, see the scripts in `/etc/rc.d/rc3.d` on an SLES system.

Figure 5-80 schematically describes the boot process of System p servers.

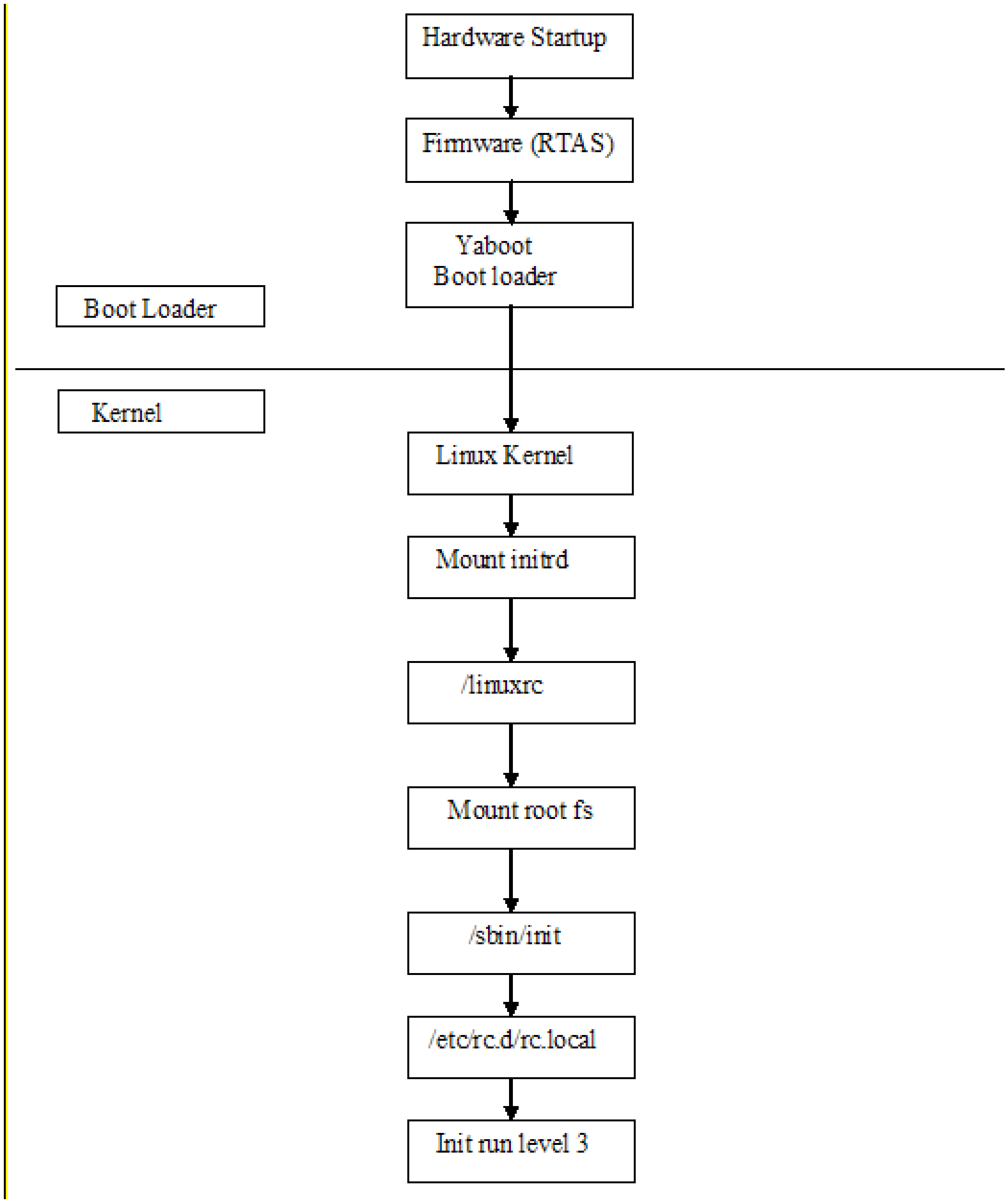


Figure 5-80: System p SLES boot sequence

5.10.4 System p in LPAR

SLES runs in a logical partition on an System p system. The hypervisor program creates logical partitions, which interacts with actual hardware and provides virtual versions of hardware to operating systems running in different logical partitions. As part of an Initial Program Load, the hypervisor performs certain initializations, listed below, before handing control over to the operating system.

5.10.4.1 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. The hypervisor assigns memory to the partition as a 64 MB contiguous load area and the balance in 256 KB chunks.
2. The boot loader loads the SLES kernel into the load area.
3. Provides system configuration data to the SLES kernel via several data areas provided within the kernel.
4. Sets up hardware translations to the SLES kernel space address 0xc000 for the first 32 MB of the load area.
5. Gives control to the SLES kernel with relocation enabled.
6. Builds the msChunks array to map the kernel view of real addresses to the actual hardware addresses.
7. Builds an event queue, which is used by the hypervisor to communicate I/O interrupts to the partition.
8. Opens a connection to a hosting partition through the hypervisor to perform any virtual I/O.
9. Starts kernel initialization by invoking `start_kernel()`.
10. `start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures, IDT tables, slab allocator (described in Section 5.5.3.6), system date, and system time.
11. Uncompresses the system `initrd` initial RAM file, mounts it, and then executes `/linuxrc`.
12. Unmount `initrd`, mounts the root file system, and executes `/sbin/init`. Resets the pid table to assign process ID one to the `init` process.
13. `/sbin/init` determines the default run level from `/etc/inittab` and performs the following basic system initialization by executing the script `/etc/rc.d/rc.sysinit`:
    1. Allows an administrator to perform interactive debugging of the startup process by executing the `/etc/sysconfig/init` script.
    2. Mounts the `/proc` special file system.
    3. Mounts the `/dev/pts` special file system.
    4. Executes `/etc/rc.d/rc.local`, which was set by an administrator to perform site-specific setup functions.
    5. Performs run-level specific initialization by executing startup scripts defined in `/etc/inittab`. The scripts are named `/etc/rc.d/rcX.d`, where X is the default run level. The default run level for a typical SLES system is 3. The following lists some of the initializations performed at run level 3.
        - Saves and restores the system entropy tool for higher quality random number generation.
        - Configures network interfaces.
        - Starts the system logging daemons.
        - Starts the `sshd` daemon.
        - Starts the `cron` daemon.
        - Probes hardware for setup and configuration.

- Starts the `agetty` program.

For more details about services started at run level 3, see the scripts in `/etc/rc.d/rc3.d` on a SLES system.

Figure 5-81 schematically describes the boot process of System p LPARs.

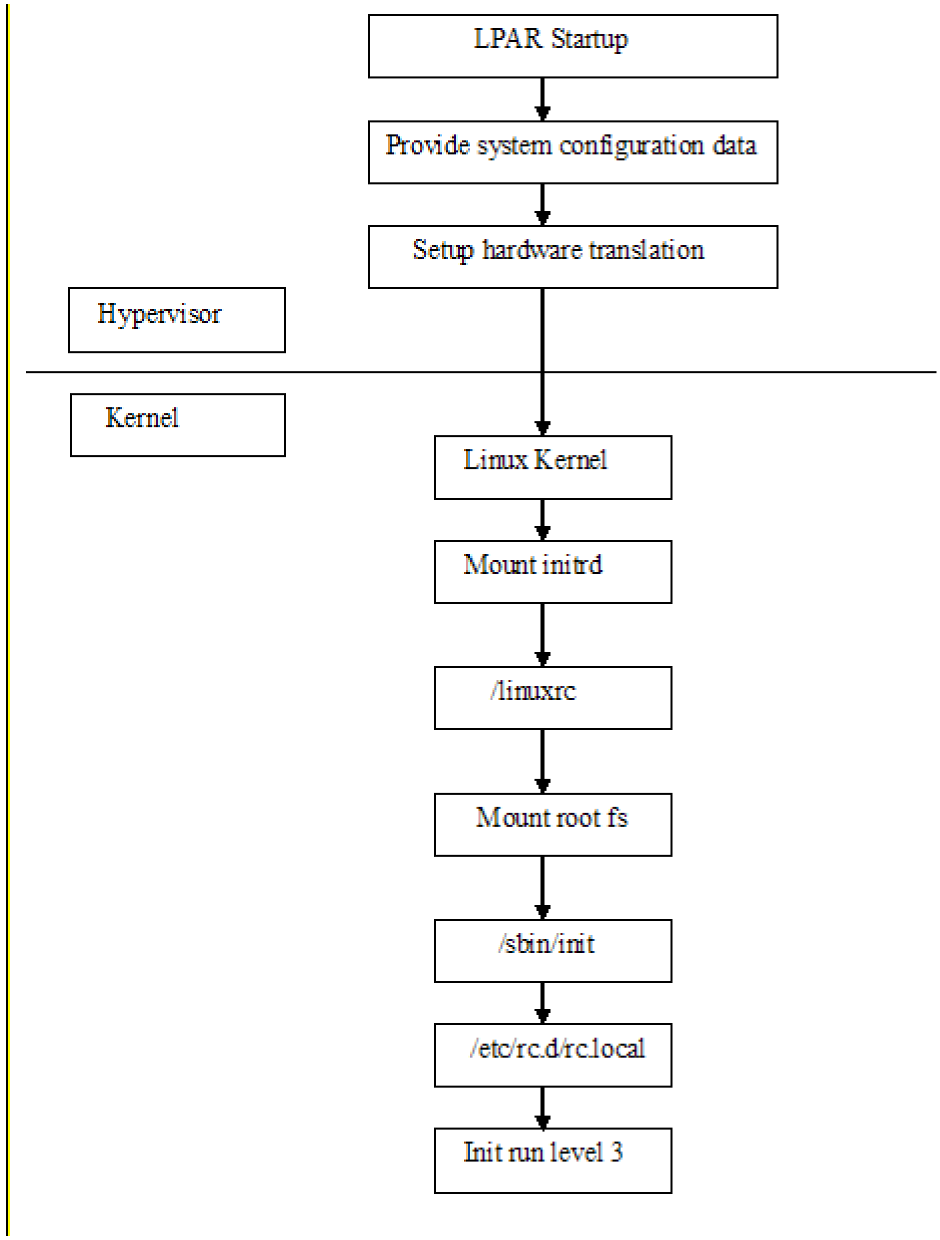


Figure 5-81: System p LPAR SLES boot sequence

5.10.5 System z

This section briefly describes the system initialization process for System z servers.

5.10.5.1 Boot methods

Linux on System z supports three installation methods: native, LPAR, and z/VM guest installations. SLES only supports z/VM guest installation. The process described below corresponds to the z/VM guest mode. The boot method for the SLES guest partition involves issuing an Initial Program Load (IPL) instruction to the Control Program (CP), which loads the kernel image from a virtual disk (DASD) device. The zipl Linux utility is responsible for creating the boot record used during the IPL process.

5.10.5.2 Control program

On a System z system, SLES runs in as a guest of the z/VM operating system. The control program, which is the System z hypervisor, interacts with real hardware and provides SLES with the same interfaces that real hardware would provide. As part of the IPL, the control program performs initializations before handing control over to the operating system.

5.10.5.3 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. For an individual SLES guest partition, on issuing an IPL instruction, the CP reads the boot record written to the DASD virtual disk by the `zipl` utility.
2. Based on the boot record, CP loads the SLES kernel image into memory and jumps to the initialization routine, handing control over to the SLES OS code.
3. `zipl` auto detects all the devices attached to the system.
4. `zipl` obtains information about the CPUs.
5. `zipl` obtains information about disk devices and disk geometry.
6. `zipl` obtains information about network devices.
7. `zipl` jumps to the `start_kernel()` function to continue kernel data structure initialization.
8. `start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures, IDT tables, slab allocator (described in Section 5.5.3.6), system date, and system time.
9. `zipl` uncompresses the initrd initial RAM file system, mounts it, and then executes `/linuxrc`.
10. `zipl` unmounts `initrd`, mounts the root file system, and executes `/sbin/init`. `zipl` resets the pid table to assign process ID one to the `init` process.
11. `/sbin/init` determines the default run level from `/etc/inittab` and performs the following basic system initialization by executing the script `/etc/rc.d/sysyinit`.
    1. Allows an administrator to perform interactive debugging of the startup process by executing the `/etc/sysconfig/init` script.
    2. Mounts the `/proc` special file system.
    3. Mounts the `/dev/pts` special file system.

4. Executes `/etc/rc.d/rc.local`, which was set by an administrator to perform site-specific setup functions.
5. Performs run-level specific initialization by executing startup scripts defined in `/etc/inittab`. The scripts are named `/etc/rc.d/rcX.d`, where X is the default run level. The default run level for a SLES system in the evaluated configuration is 3. The following lists some of the initializations performed at run level 3.
   - Saves and restores the system entropy tool for higher quality random number generation.
   - Configures network interfaces.
   - Starts the system logging daemons.
   - Starts the `sshd` daemon.
   - Starts the `cron` daemon.
   - Probes hardware for setup and configuration.
   - Starts the `agetty` program.

   For more details on services started at run level 3, see the scripts in `/etc/rc.d/rc3.d` on an SLES system.

Figure 5-82 describes the boot process for SLES as a z/VM guest.

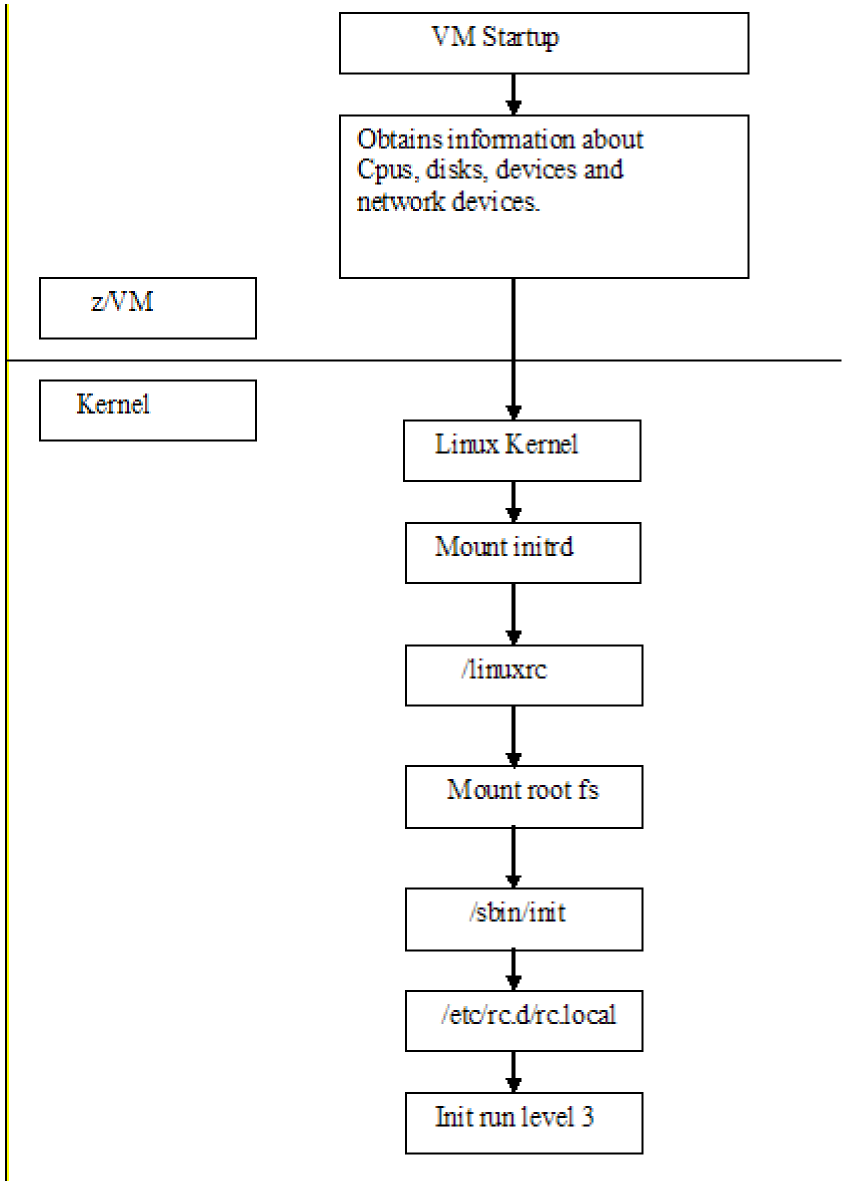


Figure 5-82: System z SLES boot sequence

5.10.6 eServer 326

This section briefly describes the system initialization process for eServer 326 servers. For detailed information on system initialization, see *AMD64 Architecture, Programmer's Manual Volume 2: System Programming*, at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf.

5.10.6.1 Boot methods

SLES supports booting from a hard disk, a CD-ROM, or a floppy disk. CD-ROM and floppy disk boots are used for installation and to perform diagnostics and maintenance. A typical boot is from a boot image on the local hard disk.

5.10.6.2 Boot loader

After the system completes the hardware diagnostics setup in the firmware, the first program that runs is the boot loader. The boot loader is responsible for copying the boot image from hard disk and then transferring control to it. SLES supports GRUB, which lets you set pointers in the boot sector to the kernel image and to the RAM file system image.

5.10.6.3 Boot process

The boot process consists of the following steps when the CPU is powered on or reset:

1. BIOS probes hardware, establishes which devices are present, and runs Power-On Self Test (POST). BIOS, is not part of TOE.
2. BIOS initializes hardware devices and makes sure they operate without IRQ or I/O port conflicts.
3. BIOS searches for the operating system to boot in an order predefined by the BIOS setting. Once a valid device is found, BIOS copies the contents of its first sector containing the boot loader into RAM and starts executing the code just copied.
4. The boot loader is invoked by BIOS to load the kernel and the initial RAM file system into the system's RAM. It then jumps to the `setup()` code.
5. The `setup()` function reinitializes the hardware devices in the computer and sets the environment for the execution of the kernel program. The `setup()` function initializes and configures hardware devices, such as the keyboard, video card, disk controller, and floating point unit.
6. The boot loader reprograms the Programmable Interrupt Controller and maps the 16 hardware interrupts to the range of vectors from 32 to 47. The boot loader switches the CPU from Real Mode to Protected Mode and then jumps to the `startup_32()` function.
7. The boot loader initializes the segmentation registers and provisional stack. It fills the area of uninitialized data of the kernel with zeros.
8. The boot loader decompresses the kernel, moves it into its final position at 0x00100000, and jumps to that address.
9. The boot loader calls the `second startup_32()` function to set the execution environment for process 0.
10. The boot loader prepares to enable long mode by enabling Physical Address Extensions and Page Global Enable.
11. The boot loader sets early boot stage 4 level page tables, enables paging, and Opteron long mode. It jumps to `reach_compatibility_mode()`, loads GDT with 64-bit segment and starts operating in 64-bit mode.
12. Initializes the segmentation registers.
13. Sets up the kernel mode stack for process 0.
14. Fills the `bss` segment of the kernel with zeros.
15. Sets up the IDT with null interrupt handlers. Puts the system parameters obtained from the BIOS and the parameters passed to the operating system into the first page frame.
16. Identifies the model of the processor. Loads `gdtr` and `idtr` registers with the addresses of the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) and jumps to the `x86_64_start_kernel()` function.

17. `x86_64_start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures IDT tables, slab allocator (described in Section 5.5.3.6), system date, and system time.
18. Uncompress the `initrd` initial RAM file system, mounts it, and then executes `/linuxrc`.
19. Unmounts `initrd`, mounts the root file system, and executes `/sbin/init`. Resets the pid table to assign process ID one to the `init` process.
20. `/sbin/init` determines the default run level from `/etc/inittab` and performs the following basic system initialization by executing the script `/etc/rc.d/rc.sysinit`.
    1. Allows an administrator to perform interactive debugging of the startup process by executing the `/etc/sysconfig/init` script.
    2. Mounts the `/proc` special file system.
    3. Mounts the `/dev/pts` special file system.
    4. Executes `/etc/rc.d/rc.local`, which was set by an administrator to perform site-specific setup functions.
    5. Performs run-level specific initialization by executing startup scripts defined in `/etc/inittab`. The scripts are named `/etc/rc.d/rcX.d`, where X is the default run level. The default run level for a typical SLES system is 3. The following lists some of the initializations performed at run level 3.
        - Saves and restores the system entropy tool for higher quality random number generation.
        - Configures network interfaces.
        - Starts the system logging daemons.
        - Starts the `sshd` daemon.
        - Starts the `cron` daemon.
        - Probes hardware for setup and configuration.
        - Starts the `agetty` program.

    For more details about services started at run level 3, see the scripts in `/etc/rc.d/rc3.d` on an SLES system.

For detailed information about the boot process, see http://tldp.org/HOWTO/Linux-Init-HOWTO.html#toc1.

Figure 5-83 schematically describes the boot process of eServer 326.

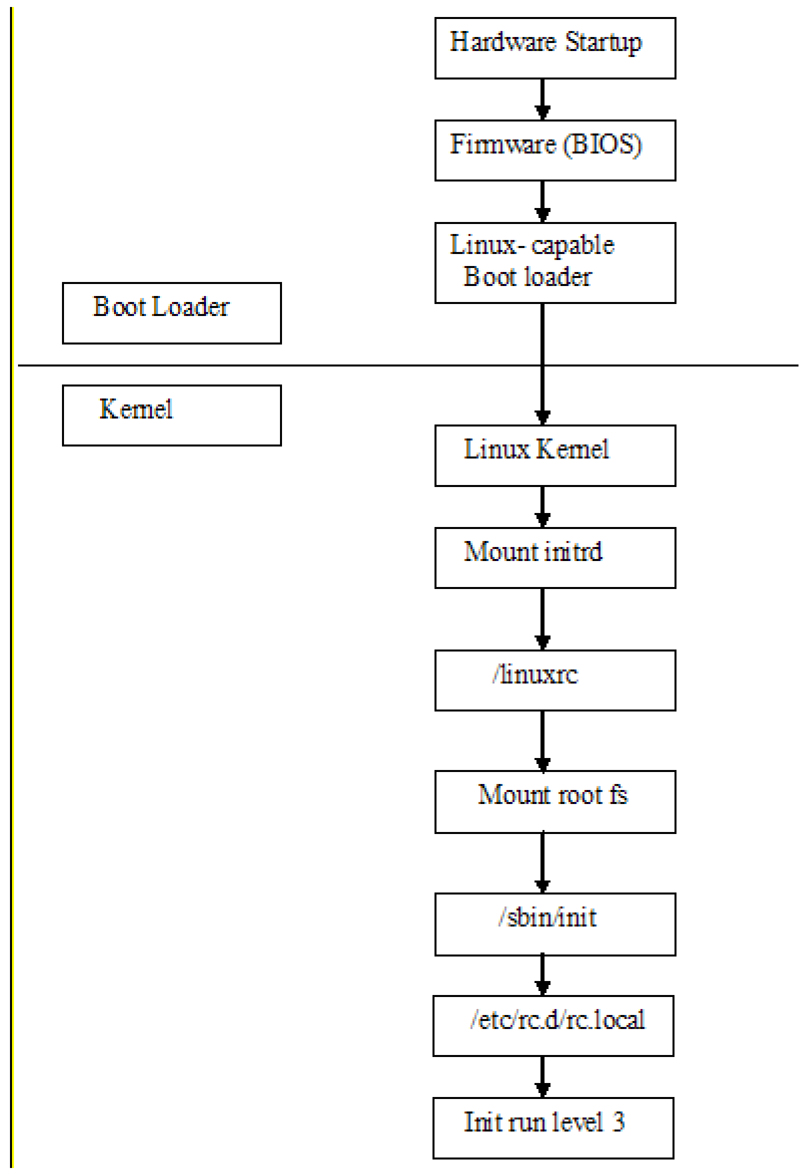


Figure 5-83: eServer 326 SLES boot sequence

5.11 Identification and authentication

Identification is when a user possesses an identity to a system in the form of a login ID. Identification establishes user accountability and access restrictions for actions on the system. Authentication is verification that the user's claimed identity is valid, and is implemented through a user password at login time.

All discretionary access-control decisions made by the kernel are based on the process's user ID established at login time and all mandatory access control decisions made by the kernel are based on the process domain established through login, which make the authentication process a critical component of a system.

The TOE implements identification and authentication through a set of trusted programs and protected databases. These trusted programs use an authentication infrastructure called the Pluggable Authentication Module (PAM). PAM allows different trusted programs to follow a consistent authentication policy. PAM provides a way to develop programs that are independent of the authentication scheme. These programs need authentication modules to be attached to them at run-time in order to work. Which authentication module is to be attached is dependent upon the local system setup and is at the discretion of the local system administrator.

This section briefly describes PAM, protected databases and their functions, trusted programs and their high level design implementation, and interaction of the identification and authentication subsystem with audit. For more detailed information, see *Linux System Security*, 2nd Edition, by Scott Mann, Ellen Mitchell and Michell Krell; and, the Linux Security HOWTO at http://www.tldp.org/HOWTO/Security-HOWTO/index.html by Kevin Fenzi and Dave Wreski.

5.11.1 Pluggable Authentication Module

PAM is responsible for the identification and authentication subsystem. PAM provides a centralized mechanism for authenticating all services. PAM allows for limits on access to applications and alternate, configurable authentication methods. For more detailed information about PAM, see the PAM project Web site at http://www.kernel.org/pub/linux/libs/pam.

5.11.1.1 Overview

PAM consists of a set of shared library modules, which provide appropriate authentication and audit services to an application. Applications are updated to offload their authentication and audit code to PAM, which allows the system to enforce a consistent identification and authentication policy, as well as generate appropriate audit records. The following trusted programs are enhanced to use PAM:

- login
- passwd
- su
- useradd, usermod, userdel
- groupadd, groupmod, groupdelsshd
- vsftpd
- chage
- chfn
- chsh

A PAM-aware application generally goes through the following steps:

1. The application makes a call to PAM to initialize certain data structures.
2. The PAM module locates the configuration file for that application from `/etc/pam.d/`*`application_name`* and obtains a list of PAM modules necessary for servicing that application. If it cannot find an application-specific configuration file, then it uses `/etc/pam.d/common-*`.
3. Depending on the order specified in the configuration file, PAM loads the appropriate modules. Refer to Section 5.16 for the mechanics of loading a shared library.
4. The `pam_loginuid.so` object associates the login uid with the login session.
5. The authentication module code performs the authentication, which, depending on the type of authentication, may require input from the user.

6. Each authentication module performs its action and relays the result back to the application.
7. The PAM library is modified to create a USER_AUTH type of audit record to note the success or failure from the authentication module.
8. The application takes appropriate action based on the aggregate results from all authentication modules.

5.11.1.2 Configuration terminology

PAM configuration files are stored in /etc/pam.d. Each application is configured with a file of its own in the /etc/pam.d directory. For example, the login configuration file is /etc/pam.d/login, and the passwd configuration file is /etc/pam.d/passwd. Each configuration file can have four columns that correspond to the entry field's module-type, control-flag, module-path, and arguments.

1. module-type: Module types are auth, which tells the application to prompt users for their passwords to determine that they are whom they claim to be; account, which verifies various account parameters, such as password age; session, manages resources associated with a service by running specified code at the start and end of the session; and, password, which updates users' authentication tokens.
2. control-flag: Control flags specify the action to be taken based on the result of a PAM module routine. When multiple modules are stacked for an application, the control flag specifies the relative importance of the modules in the stack.

   Control flags take a value, such as required, which indicates that the module must return success for service to be granted; requisite, which is similar to required, but PAM executes the rest of the module stack before returning failures to the application; optional, which indicates that the module is not required; and, sufficient, which indicates that if the module is successful, there is no need to check other modules in the stack.
3. module_path: Module path specifies the exact path name of the shared library module, or only the name of the module in /lib/security.
4. arguments: The argument field passes arguments or options to the PAM. Arguments can take values such as debug, to generate debug output, or no_warn, to prevent the PAM from passing any warning messages to the application. On the evaluated SLES system, the md5 option allows longer passwords than the usual UNIX limit of eight characters.

5.11.1.3 Modules

SLES is configured to use the following PAM modules:

- pam_unix2.so: Supports all four module types, and provides standard password-based authentication. pam_unix2.so uses standard calls from the system libraries to retrieve and set account information as well as to perform authentication. Authentication information about SLES is obtained from the /etc/passwd and /etc/shadow files. The pam_unix2.so module is configured by the /etc/security/pam_unix2.conf file, which contains options for authentication, account management, and password management.
- pam_pwcheck.so: Checks passwords by reading /etc/login.defs and making the checks provided by the Linux shadow suite. pam_pwcheck.so is configured by the /etc/security/pam_pwcheck.conf file, which instructs it to use the cracklib library to check the strength of the password. The cracklib library uses the /usr/lib/cracklib_dict.* dictionary files to evaluate the strength of the password. pam_pwcheck.so also prevents users from reusing passwords already used before, by checking the /etc/security/opasswd file.

- pam_passwdqc.so: Performs additional password strength checks. For example, it rejects passwords such as "1qaz2wsx" that follow a pattern on the keyboard. In addition to checking regular passwords it offers support for passphrases and can provide randomly generated passwords.
- pam_env.so: Loads a configurable list of environment variables, and it is configured with the file /etc/security/pam_env.conf.
- pam_shells.so: Authentication is granted if the user's shell is listed in /etc/shells. If no shell is in /etc/passwd (empty), the /bin/sh is used. It also checks to make sure that /etc/shells is a plain file and not world-writable.
- pam_limits.so: This module imposes user limits on login. It is configured using the /etc/security/limits.conf file. Each line in this file describes a limit for a user in the form: <domain> <type> <item> <value>. No limits are imposed on UID 0 accounts.
- pam_rootok.so: This module is an authentication module that performs one task: if the id of the user is 0, then it returns PAM_SUCCESS. With the sufficient /etc/pam.conf control flag, it can be used to allow password free access to some service for root.
- pam_xauth.so: This module forwards xauth cookies from user to user. Primitive access control is provided by ~/.xauth/export in the invoking user's home directory, and ~/.xauth/import in the target user's home directory. For more information, refer to /usr/share/doc/packages/pam/modules/README.pam_xauth on an SLES system.
- pam_wheel.so: Permits root access only to members of the wheel group. By default, pam_wheel.so permits root access to the system if the applicant user is a member of the wheel group. First, the module checks for the existence of a wheel group. Otherwise, the module defines the group with group ID 0 to be the wheel group. The TOE is configured with a wheel group of GID = 10.
- pam_nologin.so: Provides standard UNIX nologin authentication. If the file /etc/nologin exists, only root is allowed to log in; other users are turned away with an error message (and the module returns PAM_AUTH_ERR or PAM_USER_UNKNOWN). All users (root or otherwise) are shown the contents of /etc/nologin.
- pam_loginuid.so: Sets the login uid for the process that was authenticated. See Section 5.6.5.
- pam_securetty.so: Provides standard UNIX securetty checking, which causes authentication for root to fail unless the calling program has set PAM_TTY to a string listed in the /etc/securetty file. For all other users, pam_securetty.so succeeds.
- pam_tally.so: Keeps track of the number of login attempts made and denies access based on the number of failed attempts, which is specified as an argument to pam_tally.so module (deny = 5). This is addressed at the account module interface. The pam_tally program allows administrative users to examine and control the pam_tally PAM module's tally file.
- pam_listfile.so: Allows the use of ACLs based on users, ttys, remote hosts, groups, and shells.
- pam_deny.so: Always returns a failure.

For detailed information about all of these modules, refer to /usr/share/doc/packages/pam/modules/README.*ModuleName* on a SLES system.

5.11.2 Protected databases

The following databases are consulted by the identification and authentication subsystem during user session initiation:

- `/etc/passwd`: For all system users, it stores the login name, user ID, primary group ID, real name, home directory, and shell. Each user's entry occupies one line, and fields are separated by a colon (:). The file is owned by the root user and root group, and its mode is 644.
- `/etc/security/opasswd`: For all system users, it stores X number of most recent passwords in order to force password change history and keep the users from alternating between the same password too frequently. (`remember = X`; this is one of the options supported by pam_unix.so.) The file is owned by the root user and root group, and its mode is 644.
- `/etc/group`: For system groups, stores group names, group IDs, supplemental group IDs, and group memberships. Each group's entry occupies one line and fields are separated by a colon (:). The file is owned by the root user and root group, and its mode is 644.
- `/etc/shadow`: For all system users, it stores the user name, hashed password, last password change time (in days since epoch), minimum number of days that must pass before password can be changed again, maximum number of days after which the password must be changed, number of days before the password expires when the user is warned, number of days after the password expires that the account is locked, and total lifetime of the account. The MD5 hashing algorithm is used to build the password checksum. The file is owned by the root user and shadow group, and its mode is 400.
- `/etc/gshadow`: This is the group counterpart of the shadow file. For all system groups, it stores group names, group hashed passwords, and membership information. The MD5 hashing algorithm is used to build the password checksum. The file is owned by the root user and shadow group, and its mode is 400. Group passwords are not used in the TOE.
- `/var/log/lastlog`: The time and date of the last successful login for each user is stored here. The file is owned by the root user and tty group, and its mode is 644.
- `/var/log/faillog`: faillog maintains the count of login failures and the limits for each user account. The file is fixed length record, indexed by numerical UID. Each record contains the count of login failures since the last successful login, the maximum number of failures before the account is disabled, the line the last login failure occurred on, and the date the last login failure occurred. The file is owned by the root user and root group, and its mode is 644.
- `/etc/login.defs`: This data base defines various configuration options for the login process, such as minimum and maximum user ID for automatic selection by the command useradd, minimum and maximum group ID for automatic selection by the command groupadd, password aging controls, default location for mail, and whether to create a home directory when creating a new user. The file is owned by the root user and root group, and its mode is 644.
- `/etc/securetty`: Lists ttys from which the root user can log in. Device names are listed one per line, without the leading /dev/. The file is owned by the root user and root group, and its mode is 644.
- `/var/run/utmp`: The utmp file stores information about who is currently using the system. The utmp file contains a sequence of entries with the name of the special file associated with the user's terminal, the user's login name, and the time of login in the form of time. The file is owned by the root user and tty group, and its mode is 664.
- `/var/log/wtmp`: The wtmp file records all logins and logouts. Its format is exactly like utmp except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name tilde (~) with a user name of "shutdown" or "reboot" indicates a system shutdown or reboot, and the pair of terminal names "|"/"}" logs the old new system time when the command date changes it. The file is owned by the root user and tty group, and its mode is 664.

- /etc/ftpusers: The ftpusers text file contains a list of users who cannot log in using the File Transfer Protocol (FTP) server daemon. The file is owned by the root user and root group, and its mode is 644.
- /etc/apparmor/* and /etc/apparmor.d/*: The directories /etc/apparmor and /etc/apparmor.d contain several configuration files that are used by the AppArmor LSM modules. Both directories are owned by the root user and root group, and their mode is 755.

5.11.2.1 Access control rules

5.11.2.1.1 DAC

Discretionary Access Checks (DAC) access control rules specify how a certain process with appropriate DAC security attributes can access an object with a set of DAC security attributes. In addition, DAC access control rules also specify how subject and object security attributes transition to new values and under what conditions. DAC access control lists are described in detail in Section 5.1.5.2.

5.11.2.1.2 Software privilege

Software privilege for DAC policy is based on the user ID of the process. At any time, each process has an effective user ID, an effective group ID, and a set of supplementary group IDs. These IDs determine the privileges of the process. A process with a user ID of 0 is a privileged process, with capabilities of bypassing the access control policies of the system. The root user name is commonly associated with user ID 0, but there can be other users with this ID.

Additionally, the SLES kernel has a framework for providing software privilege for DAC policy through capabilities. These capabilities, which are based on the POSIX.1e draft, allow breakup of the kernel software privilege associated with user ID zero into a set of discrete privileges based on the operation being attempted. For example, if a process is trying to create a device special file by invoking the mknod() system call, instead of checking to ensure that the user ID is zero, the kernel checks to ensure that the process is capable of creating device special files. In the absence of special kernel modules that define and use capabilities, as is the case with the TOE, capability checks revert back to granting kernel software privilege based on the user ID of the process.

5.11.3 Trusted commands and trusted processes

The Identification and Authentication subsystem contains the agetty and mingetty trusted processes and the gpasswd, login, passwd, and su trusted commands.

5.11.3.1 agetty

agetty, the alternative Linux getty, is invoked from /sbin/init when the system transitions from a single-user mode to a multi-user mode. agetty opens a tty port, prompts for a login name, and invokes /bin/login to authenticate. Refer to the agetty man page for more detailed information. agetty follows these steps:

1. Sets language.
2. Parses command line setup options such as timeout and the alternate login program.
3. Updates the utmp file with tty information.
4. Initializes terminal I/O characteristics. Examples are modems or regular terminals.
5. Prompts for login name.

6. Execs the `login` program.

The steps that are relevant to the identification and authorization subsystem are step 5, which prompts for the user's login name, and step 6, which executes the `login` program. The administrator can also use a command-line option to terminate the program if a user name is not entered within a specific amount of time.

5.11.3.2 gpasswd

The `gpasswd` program administers the `/etc/group` and `/etc/gshadow` files. `gpasswd` allows system administrators to designate group administrators for a particular group. Refer to the `gpasswd` man page for more detailed information. Group passwords are not used on the TOE.

5.11.3.3 login

The `login` program is used to authenticate a user signs to the TOE. If root is trying to log in, the program makes sure that the login attempt is being made from a secure terminal listed in `/etc/securetty`. `login` prompts for the password and turns off the terminal echo in order to prevent displaying the password as it is being typed by the user. `login` then verifies the password for the account. If an initial password is not set for a newly created account, the user is not allowed to log in to that account. Unsuccessful login attempts are tallied, and access is denied, if the number of failed attempts exceeds the number specified as argument to the `pam_tally.so` module (`deny=5`). Once the password is successfully verified, various password aging restrictions, which are set up in `/etc/login.defs`, are checked. If the password has expired, the login program requests the user to change his or her password. If the password age is satisfactory, the program sets the user ID and group ID of the process, changes the current directory to the user's home directory, and executes the shell specified in the `/etc/passwd` file. Refer to the login man page for more detailed information. Login generally follows these steps:

1. Sets language.
2. Parses command-line options.
3. Checks the tty name.
4. Sets the process group ID.
5. Gets control of the tty by killing processes left on this tty.
6. Calls `pam_start()` to initialize PAM data structures, including hostname and tty.
7. If a password is required and a username is not yet set, it prompts for a user name.
8. It calls `pam_authenticate()` in a loop to cycle through all configured methods. Audit records are created with the success and failure result of each configured authentication method.
9. If failed attempts exceed the maximum allowed, it exits.
10. Performs account management by calling `pam_acct_mgmt()`.
11. Sets up supplementary group list.
12. Updates the `utmp` and `wtmp` files.
13. Changes ownership of the tty to the login user. When the user logs off, the ownership of the tty reverts back to root.
14. Changes the access mode of the tty.
15. Sets the primary group ID.
16. Sets environment variables.

17. Sets effective, real, and saved user ID.
18. Changes directory to the user's home directory.
19. Executes shell.

5.11.3.4 mingetty

`mingetty`, the minimal Linux `getty`, is invoked from `/sbin/init` when the system transitions from single-user mode to multi-user mode. `mingetty` opens a pseudo tty port, prompts for a login name, and invokes `/bin/login` to authenticate. Refer to the `mingetty` man page for more detailed information. `mingetty` follows these steps:

1. Sets language.
2. Parses command-line setup options such as timeout and the alternate login program.
3. Updates the `utmp` file with pseudo tty information.
4. Prompts for login name.
5. Execs the `login` program.

The steps that are relevant to the identification and authorization subsystem are step 4, which prompts for the user's login name, and step 5, which executes the `login` program. The administrator can also use a command-line option to terminate the program if a user name is not entered within a specific amount of time.

5.11.3.5 newgrp

The `newgrp` command changes the group ID using the group password for authentication. If run with a command, it runs the command as a child shell and control returns to the parent, whose GID remains unaltered. If run without a command argument, it execs an interactive shell. If no group is specified, the GID from `/etc/passwd` is used. For more information on `newgrp`, see the newgrp(8) man page.

`newgrp` typically follows these processing steps:

1. Opens audit.
2. Sets its locale and message catalog information.
3. Gets its own name.
4. Opens its log file.
5. Gets its own gid.
6. Initializes environment variable handling.
7. Gets its own passwd entry.
8. Exits if there is no passwd entry.
9. Processes the command line.
10. Gets the group name of the group.
11. Checks all group with same GID for user membership.
12. Gets the shadow group name.
13. Check to see if the user needs to authenticate to take on the new GID.
14. Prompts for a password of authentication is necessary.
15. Logs the result.

16. Sets up signals.
17. Forks a child.
18. Parent waits on child's return; child continues:
19. Adds the new GID to the group list.
20. Sets the GID.
21. Logs an audit record.
22. Starts a shell if the `-c` flag was specified.
23. Looks for the `SHELL` environment variable or, if `SHELL` is not set defaults to `/bin/sh`.
24. Gets the `basename` of the shell for `argv[0]`.
25. Closes the password and group files.
26. Changes to home directory if doing a login.
27. Logs an audit record.
28. Execs a shell with a command.
29. Closes its log.
30. Exits.

5.11.3.6 passwd

`passwd` updates a user's authentication tokens. `passwd` is configured to work through the PAM API. `passwd` configures itself as a password service with PAM, and uses configured password modules to authenticate and then update a user's password. `passwd` turns off terminal echo, while the user is typing the old as well as the new password, in order to prevent the password from being displayed as it is being typed by the user. Refer to the `passwd` man page for more detailed information. `passwd` generally follows these steps.

1. Parses command-line arguments.
2. Handles requests for locking, unlocking, and clearing of passwords for an account.
3. If requested, displays account status.
4. If requested, updates password aging parameters
5. Reads new password from standard input.
6. Starts PAM session with a call to `pam_start()`.
7. Calls `pam_chauthtok()` to perform password strength checks and to update the password.
8. Generates audit record indicating successful update of the password.

5.11.3.7 su

`su` allows a user to switch identity. `su` changes the effective and real user and group ID to those of the new user. Refer to the `su` man page for more detailed information. `su` generally follows these steps:

1. Sets language.
2. Sets up a variable indicating whether the application user is the root user.
3. Gets current tty name for logging.

4. Processes command-line arguments.
5. Sets up the environment variable array.
6. Invokes `pam_start()` to initialize the PAM library, and to identify the application with a particular service name.
7. Invokes `pam_set_item()` to record the tty and user name.
8. Validates the user that the application invoker is trying to become.
9. Invokes `pam_authenticate()` to authenticate the application user. Terminal echo is turned off while the user is typing his or her password. Generates audit record to log the authentication attempt and its outcome.
10. Invokes `pam_acct_mgmt()` to perform module-specific account management.
11. If the application user is not root, it checks to make sure that the account permits `su`.
12. Makes new environment active.
13. Invokes `setup_groups()` to set primary and supplementary groups.
14. Invokes `pam_setcred()` to set parameters such as resource limits, console groups, and so on.
15. Becomes the new user by invoking `change_uid()`. For normal users, `change_uid()` sets the real and effective user ID. If the caller is root, real and saved user ID are set as well.

5.11.4 Interaction with audit

Trusted processes and trusted commands of the identification and authentication subsystem are responsible for setting the credentials for a process. Once a user is successfully authenticated, these trusted processes and trusted commands associate the user's identity to the processes, which are performing actions on behalf of the user.

The audit subsystem tries to record security-relevant actions performed by users. Because the user identity attributes such as uid can be changed by an appropriately privileged process, the audit subsystem in SLES provides a mechanism by which actions can be associated, irrefutably, to a login user.

This is achieved by extending the process task structure to contain a login id. This login id can only be set once, and once set cannot be changed, regardless of process privileges. Trusted processes and trusted programs that perform authentication set it. Programs such as `login`, `cron`, and `sshd`, which authenticate a user and associate a uid with the user process, set this login id to that uid corresponding to the login user.

5.12 Network applications

This section describes the network applications subsystem. The network applications subsystem contains the Secure Socket Layer (SSL) interface, and the `sshd` and `vsftpd` trusted processes, which interact with the PAM modules to perform authentication. The network application subsystem also includes the ping program. These trusted processes and trusted programs recognize different hosts in the LAN by their IP addresses or their names. Host names are associated with IP addresses using the `/etc/hosts` file.

5.12.1 OpenSSL Secure socket-layer interface

Network communications take place through well-known standards that form the network stack. While public standards allow different systems to communicate with each other, they also open up the possibility of various kinds of attacks.

Cryptography can be used to neutralize some of these attacks and to ensure confidentiality and integrity of network traffic. Cryptography can also be used to implement authentication schemes using digital signatures. The TOE supports a technology based on cryptography called OpenSSL.

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL) versions 2 and 3, and Transport Layer Security (TLS) version 1 network protocols and related cryptography standards required by them.

SSL, which is encryption-based, is a technology that provides message encryption, server authentication, message integrity, and optional client authentication. This section briefly describes the SSL protocol and how it is used to provide secure communication to and from an SLES system. For more detailed information about SSL, refer to the following:

- Open SSL Web site at http://www.openssl.org/docs.
- IBM Redbook *TCP/IP Tutorial and Technical Overview*, by Adolfo Rodriguez, et al. at http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf.
- "The TLS Protocol version 1.1" by Tim Dierks and Eric Rescorla at http://www.ietf.org/rfc/rfc2246.txt?number=2246.
- *Internet Security Protocols: SSLeay & TLS*, by Eric Young.
- *Cryptography and Network Security Principles and Practice*, 2nd Edition, by William Stallings.

SSL was originally designed by Netscape. SSL version 3 was designed with public input. As SSL gained in popularity, a Transport Layer Security (TLS) working group was formed to submit the protocol for Internet standardization. OpenSSL implements Secure Socket Layer (SSL versions 2 and 3) and Transport Layer Security (TLS version 1) protocols, as well as a full-strength general purpose cryptography library. Because TLS is based on SSL, the rest of this section uses the term SSL to describe both the SSL and TLS protocols. Where the protocols differ, TLS protocols are identified appropriately.

SSL is a socket-layer security protocol that is implemented at the transport layer. SSL is a reliable connection-based protocol and therefore available on top of TCP but not UDP.

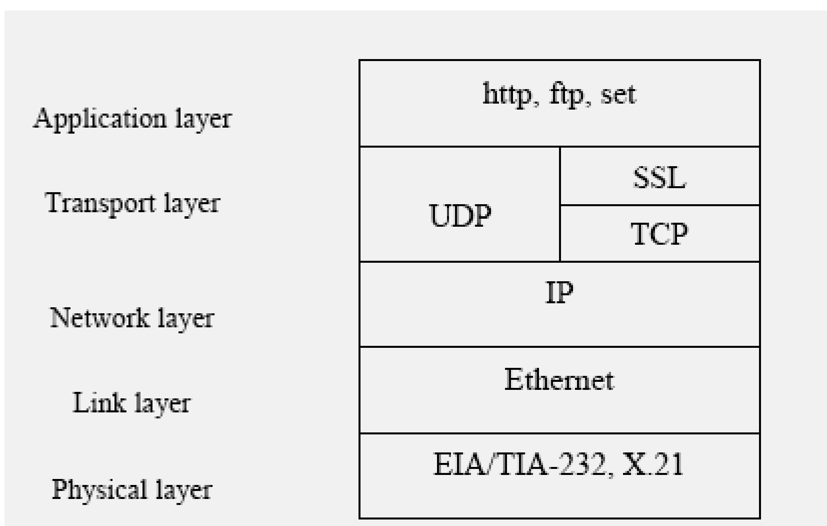


Figure 5-84: SSL location in the network stack

5.12.1.1 Concepts

SSL is used to authenticate endpoints and to secure the contents of the application-level communication. An SSL-secured connection begins by establishing the identities of the peers, and establishing an encryption method and key in a secure way. Application-level communication can then begin. All incoming traffic is decrypted by the intermediate SSL layer and then forwarded on to the application; outgoing traffic is encrypted by the SSL layer before transmission.

SSL uses encryption with symmetric keys for data transfer, encryption with asymmetric keys for exchanging symmetric keys, and one-way hash functions for data integrity. The following sections briefly describe encryption and message-digest concepts, and how they are used to implement data confidentiality, data integrity, and the authentication mechanism.

5.12.1.1.1 Encryption

Encryption is a process of disguising a message. Encryption transforms a clear-text message into cipher text.

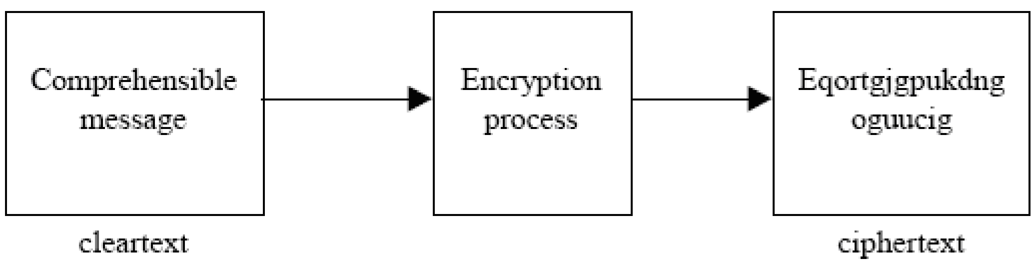


Figure 5-85: Encryption

Decryption converts cipher text back into the original, comprehensible clear text.

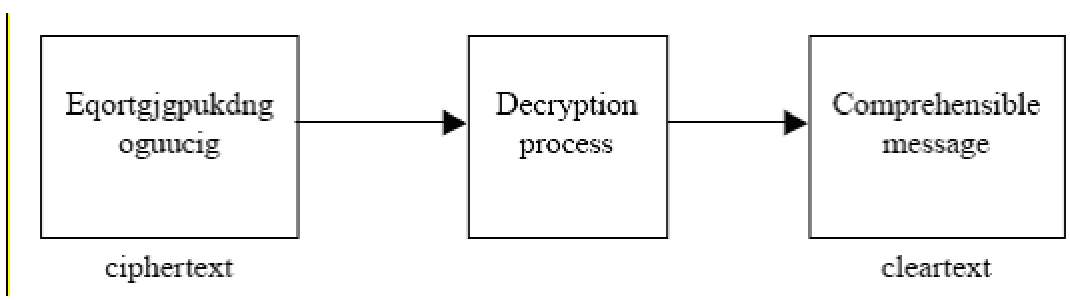


Figure 5-86: Decryption

Most encryption processes involve the use of an algorithm and a key. For example, in the previous illustration, the algorithm was “replace alphabets by moving forward” and the key was 2.

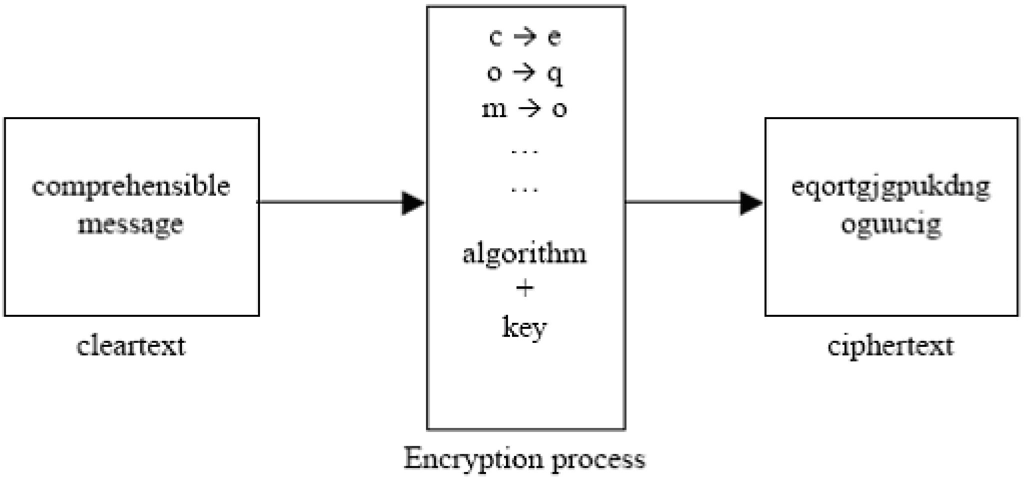


Figure 5-87: Encryption Algorithm and Key

Data confidentiality can be maintained by keeping the algorithm, the key, or both, secret from unauthorized people. In most cases, including OpenSSL, the algorithm used is well-known, but the key is protected from unauthorized people.

5.12.1.1.1.1 Encryption with symmetric keys

A symmetric key, also known as a secret key, is a single key that is used for both encryption and decryption. For example, key = 2 used in the above illustration is a symmetric key. Only the parties exchanging secret messages have access to this symmetric key.

5.12.1.1.1.2 Encryption with asymmetric keys

Asymmetric key encryption and decryption, also known as public key cryptography, involve the use of a key pair. Encryption performed with one of the keys of the key pair can only be decrypted with the other key of the key pair. The two keys of the key pair are known as public key and private key. A user generates public and private keys from a key pair. The user then makes the public key available to others while keeping the private key a secret.

Figure 5-88 conceptually illustrates the creation of asymmetric keys for encryption and decryption.

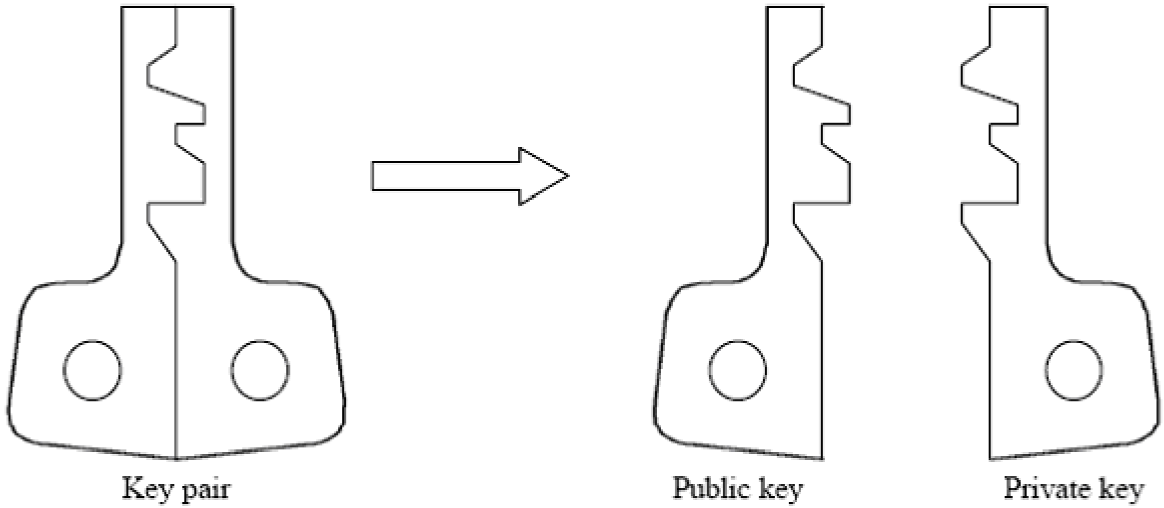


Figure 5-88: Asymmetric keys

If encryption is done with a public key, only the corresponding private key can be used for decryption. This allows a user to communicate confidentially with another user by encrypting messages with the intended receiver's public key. Even if messages are intercepted by a third party, the third party cannot decrypt them. Only the intended receiver can decrypt messages with his or her private key. The following diagram conceptually illustrates encryption with a public key to provide confidentiality.

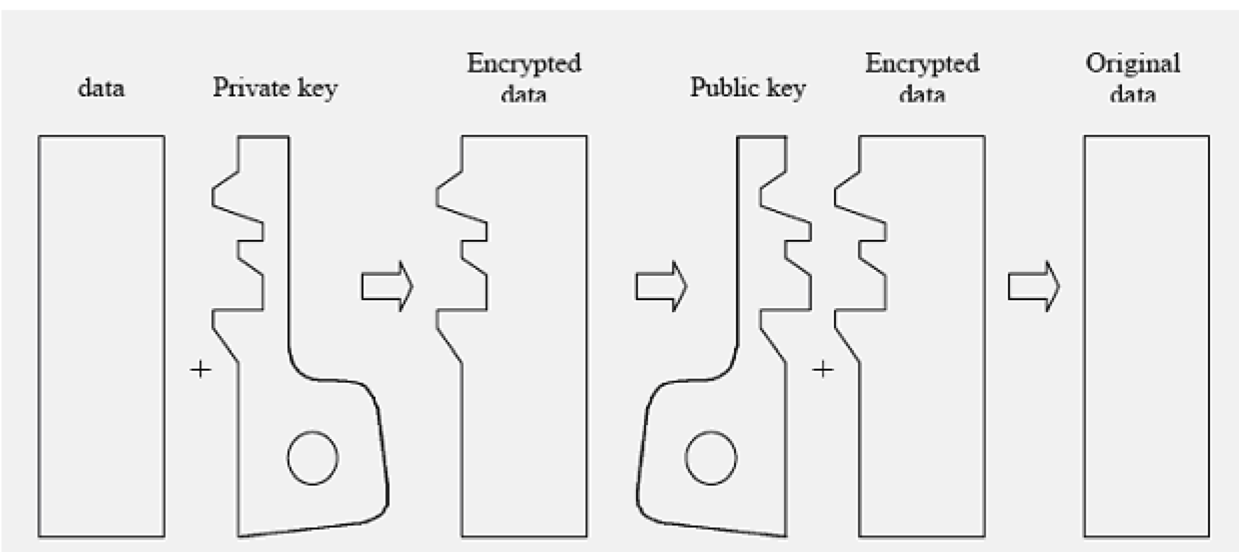


Figure 5-89: Encryption with private key provides authentication

5.12.1.1.2 Message digest

A message digest is text in the form of a single string of digits created with a one-way hash function. One-way hash functions are algorithms that transform a message of arbitrary length into a fixed length tag called a message digest.

A good hash function can detect even a small change in the original message to generate a different message digest. The hash function is one-way; it is not possible to deduce the original message from its message digest.

Message digests are used to provide assurance of message integrity. The sender generates a message digest for each of the messages being sent. Each message is transmitted, along with its message digest. The receiver separates the message digest from the message, generates a new message digest from the received message using the same algorithm used by the sender, and compares the received message digest with the newly generated one.

If the two message digests are different, then the message was altered on the way. If the two message digests are identical, then the receiver can be assured that the message's integrity was not compromised during transmission.

5.12.1.1.3 Message Authentication Code (MAC)

A message authentication code (MAC) is a type of message digest that is created by encrypting the output of a one-way hash function with a symmetric key.

5.12.1.1.4 Digital certificates and certificate authority

Cryptography with an asymmetric key depends on public keys being authentic. If two people are exchanging their public keys over an untrusted network, then that process introduces a security vulnerability. Intruders can intercept messages between them, replace their public keys with their own public keys, and monitor their network traffic. The solution for this vulnerability is the digital certificate. A digital certificate is a file that ties an identity to the associated public key.

This association of identity to a public key is validated by a trusted third party known as the certificate authority. The certificate authority signs the digital certificate with its private key. In addition to a public key and an identity, a digital certificate contains the date of issue and expiration date. OpenSSL supports the international standard, ISO X.509, for digital certificates.

5.12.1.2 SSL architecture

SSL occupies a space between the transport and application layer in the network stack, and consists of two layers. Both layers use services provided by the layer below them to provide functionality to the layers above them. The lower layer consists of the SSL Record Protocol, which uses symmetric key encryption to provide confidentiality to data communications over a reliable, connection-oriented, transport protocol TCP. The upper layer of SSL consists of the SSL Handshake Protocol, the SSL Change Cipher Spec Protocol, and the SSL Alert Protocol.

The SSL Handshake Protocol is used by the client and server to authenticate each other, and to agree on encryption and hash algorithms to be used by the SSL Record Protocol. The authentication method supported by SSL in the evaluated configuration is client and server authentication using X.509 certificates.

The SSL Change Cipher Spec changes the Cipher suite of encryption and hash algorithms used by the connection. The SSL Alert Protocol reports SSL-related errors to communicating peers.

Figure 5-90 depicts different SSL protocols and their relative positions in the network stack.

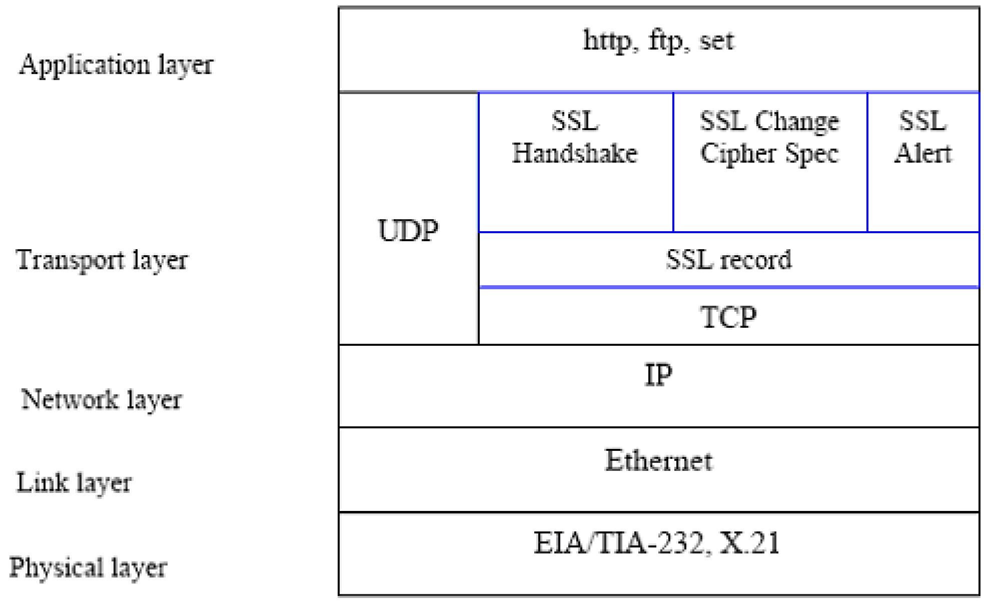


Figure 5-90: SSL Protocol

The SSL architecture differentiates between an SSL session and an SSL connection. A connection is a transient transport device between peers.

A session is an association between a client and a server. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of security parameters for each new connection. A session is identified with a session identifier, peer certificate, compression method, cipher spec, master secret, and is_resumable flag. A connection is identified with a server and client random numbers, a server write MAC secret key, a client write MAC secret key, a server write key, a client write key, initialization vectors, and sequence numbers.

5.12.1.2.1 SSL handshake protocol

The SSL handshake protocol is responsible for performing authentication of peers that are attempting secure communications. The SSL handshake protocol negotiates security parameters (encryption and hash algorithms) to be used by the SSL record protocol, and exchanges PreMasterSecret, which is used to generate authentication and encryption keys.

The handshake protocol is the most complex part of SSL. It starts with mandatory authentication of the server. Client authentication is optional. After successful authentication, the negotiation for the cipher suite, with the encryption algorithm, MAC algorithm, and cryptographic keys, takes place. Security parameters, set up by the handshake protocol, are used for all connections in a session. The following diagram illustrates the handshake protocol.

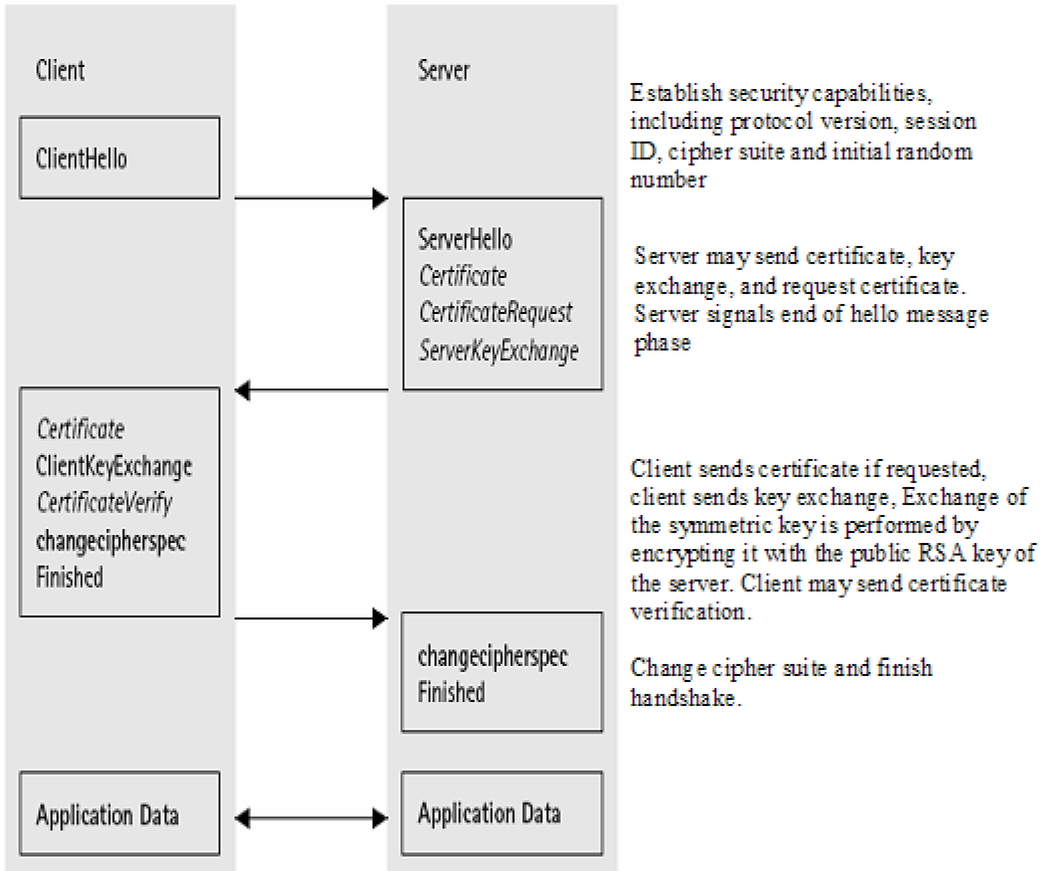


Figure 5-91: Handshake protocol (optional or content-dependent handshake messages are in italic type)

1. Client hello message: The CipherSuite list, passed from the client to the server in the client hello message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (first choice first). Each CipherSuite defines both a key exchange algorithm and a CipherSpec. The server selects a cipher suite or, if no acceptable choices are presented, returns a handshake failure alert and closes the connection.
2. Server key exchange message: The server key exchange message is sent by the server if it has no certificate, has a certificate only used for signing (that is, DSS [DSS] certificates, or signing-only RSA [RSA] certificates), or FORTEZZA KEA key exchange. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.
3. Client key exchange message (RSA encrypted premaster secret message): In the evaluated configuration, RSA is used for key agreement and authentication. The client generates a 48-byte pre-master secret, encrypts it under the public key from the server's certificate or temporary RSA key from a server key exchange message, and sends the result in an encrypted premaster secret message.
4. Certificate verify message: This message is used to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (that is, all certificates except those containing fixed Diffie-Hellman parameters).

For the list of Cipher suites supported, see FCS_COP.1(2) in the Security Target.

5. SSL Change cipher spec protocol: The SSL change cipher spec protocol signals transitions in the security parameters. The protocol consists of a single message, which is encrypted with the current security parameters. Using the change cipher spec message, security parameters can be changed by either the client or the server. The receiver of the change cipher spec message informs the SSL record protocol of the updates to security parameters.
6. SSL alert protocol: The SSL alert protocol communicates SSL-specific errors, such as errors encountered during handshake or message verification, to the appropriate peer.
7. SSL record protocol: The SSL record protocol takes messages to be transmitted, fragments them into manageable blocks, and optionally compresses them. Then, using all the negotiated security parameters, applies a message authentication code (MAC), encrypts the data, and transmits the result to the transport layer (TCP). The received data is decrypted, verified, decompressed, and reassembled. It is then delivered to a higher layer.

   The SSL record protocol provides confidentiality by encrypting the message with the shared secret key negotiated by the handshake protocol. The SSL record protocol provides message integrity by attaching a MAC to the message. The MAC is created with another shared secret key negotiated by the handshake protocol.

Figure 5-92 [STALLS] depicts the operation of the SSL record protocol.

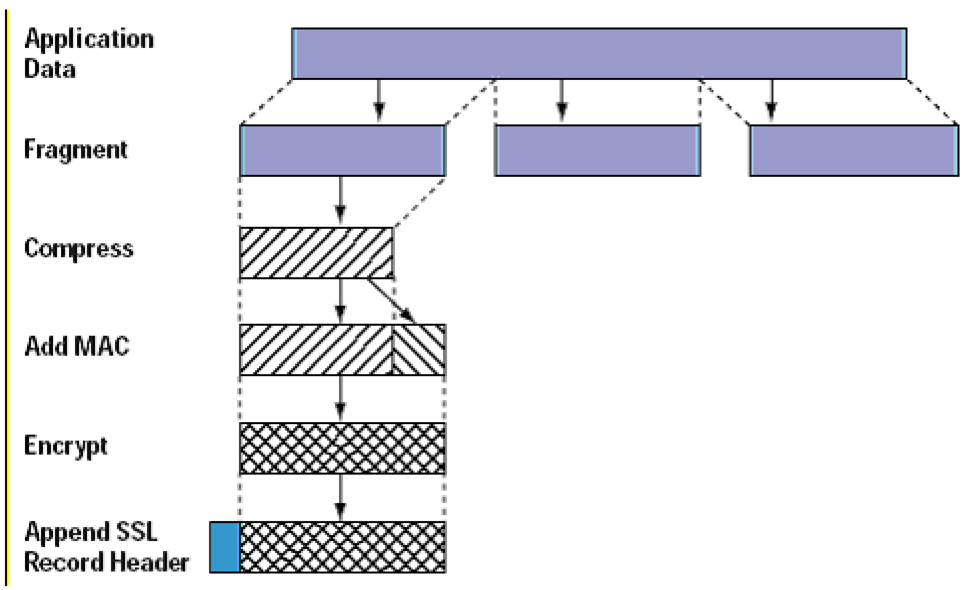


Figure 5-92: SSL protocol action

5.12.1.3 OpenSSL algorithms

This section briefly describes various encryption and hash algorithms supported by OpenSSL on the TOE.

5.12.1.4 Symmetric ciphers

OpenSSL on the TOE supports the following symmetric key encryption algorithms. For a detailed description of each of these algorithms, refer to their man pages.

- Blowfish: Blowfish is a block cipher that operates on 64-bit blocks of data. It supports variable key sizes, but generally uses 128-bit keys.
- Data Encryption Standard (DES): DES is a symmetric key cryptosystem derived from the Lucifer algorithm developed at IBM. DES describes the Data Encryption Algorithm (DEA). DEA operates on a 64-bit block size and uses a 56-bit key.
- TDES (3DES): TDES, or Triple DES, encrypts a message three times using DES. This encryption can be accomplished in several ways. For example, using two keys, the message can be encrypted with key 1, decrypted with key 2, and encrypted again with key 1. With three keys, the message can be encrypted 3 times with each encryption using a different key.
- International Data Encryption Algorithm (IDEA): The IDEA cipher is secret key block encryption algorithm developed by James Massey and Xuejia Lai. IDEA operates on 64-bit plain text blocks and uses a 128-bit key.
- RC4: RC4, proprietary of RSA Security Inc., is a stream cipher with a variable key length. A typical key length of 128-bit is used for strong encryption.
- RC5: RC5 is a cryptographic algorithm invented by Ronald Rivest of RSA Security Inc. RC5 is a block cipher of variable block length and encrypts through integer addition, the application of a bit-wise eXclusive OR, and variable rotations. The key size and number of rounds are also variable.
- Advanced encryption standard (AES): AES is a cryptographic algorithm created by researchers Joan Daemen and Vincent Rijmen. AES is an iterative, symmetric-key block cipher that can use keys of 128, 192, and 256 bits, and encrypts and decrypts data in blocks of 128 bits (16 bytes).

5.12.1.4.1 Asymmetric ciphers

OpenSSL on the TOE supports the following asymmetric key encryption algorithms. For a detailed description of each of these algorithms, refer to their man pages.

- Digital Signature Algorithm (DSA): DSA is based on a modification to the El Gamal digital signature methodology, which is based on discrete logarithms. DSA conforms to US Federal Information Processing Standard FIPS 186, and ANSI X9.30.
- Diffie-Hellman: The Diffie-Hellman Key Exchange is a method for exchanging secret keys over a non-secure medium, without exposing the keys.
- RSA: RSA, derived from the last names of its inventors, Rivest, Shamir, and Addleman, is a public key crypto system, which is based on the difficulty of factoring a number that is the product of two large prime numbers.

5.12.1.4.2 Certificates

OpenSSL on the TOE supports the X.509 certificate format. For a detailed description of this format, refer to its manual page.

The X.509 certificate is a structured grouping of information. X.509 contains subject information, the public key of the subject, the name of the issuer, and the active key lifetime. An X.509 certificate is digitally signed by the certificate authority.

5.12.1.4.3 Hash functions

OpenSSL on the TOE supports the following hash functions to generate message authentication codes. For a detailed description of each of these functions, refer to their manual pages.

MD2, MD4, and MD5 are cryptographic message-digest algorithms that take a message of arbitrary length and generate a 128-bit message digest. In MD5, the message is processed in 512-bit blocks in four distinct rounds.

MDC2 is a method to construct hash functions with 128-bit output from block ciphers. These functions are an implementation of MDC2 with DES.

RIPEMD is a cryptographic hash function with 160-bit output.

The Secure Hash Algorithm (SHA) is a cryptographic hash function with 160-bit output. It is defined in the Federal Information Processing Standard - FIPS 180. SHA-1 sequentially processes blocks of 512 bits when computing a message digest.

5.12.2 Secure Shell

Secure Shell (SSH) is a network protocol that provides a replacement for insecure remote login and command execution facilities such as `telnet`, `rlogin`, and Remote Shell (`rsh`). SSH encrypts traffic, preventing traffic sniffing and password theft.

On a local system, the user starts the SSH client to open a connection to a remote server running the `sshd` daemon. If the user is authenticated successfully, an interactive session is initiated, allowing the user to run commands on the remote system. SSH is not a shell in the sense of a command interpreter, but it permits the use of a shell on the remote system.

In addition to interactive logins, the user can tunnel TCP network connections through the existing channel, allowing the use of X11 and other network-based applications, and copy files through the use of the `scp` and `sftp` tools. OpenSSH is configured to use the PAM framework for authentication, authorization, account maintenance, and session maintenance. Password expiration and locking are handled through the appropriate PAM functions.

Communication between the SSH client and SSH server uses the SSH protocol, version 2.0. The SSH protocol requires that each host have a host specific key. When the SSH client initiates a connection, the keys are exchanged using the Diffe-Hellman protocol. A session key is generated, and all traffic is encrypted using this session key and the agreed-upon algorithm.

Default encryption algorithms supported by SSH are 3DES (triple DES) and blowfish. The default can be overridden by providing the list in the server configuration file with the “ciphers” keyword.

The default message authentication code algorithms supported by SSH are SHA-1 and MD5. The default can be overridden by providing the list in the server configuration file with the keyword MACs. Refer to Section 5.12.1.4.3 of this document for brief descriptions of these algorithms.

Encryption is provided by the OpenSSL package, which is a separate software package. The following briefly describes the default SSH setup with respect to encryption, integrity check, certificate format, and key exchange protocol.

- Encryption: The default cipher used by SSH is 3des-cbc (three-key 3DES in CBC mode). The 3des-cbc cipher is three key triple-DES (encrypt-decrypt-encrypt), where the first 8 bytes of the key are used for the first encryption, the next 8 bytes for the decryption, and the following 8 bytes for the final encryption. This requires 24 bytes of key data, of which 168 bits are actually used. To implement CBC mode, outer chaining must be used. That is, there is only one initialization vector. This is a block cipher with 8 byte blocks. This algorithm is defined in [SCHNEIR].
- Integrity check: Data integrity is protected by including a message authentication code (MAC) with each packet that is computed from a shared secret, packet sequence number, and the contents of the packet. The message authentication algorithm and key are negotiated during key exchange. Initially, no MAC will be in effect, and its length must be zero. After key exchange, the selected MAC will be computed before encryption from the concatenation of packet data:

mac = MAC (key, sequence_number || unencrypted_packet)

where unencrypted_packet is the entire packet without MAC (the length fields, payload and padding), and sequence_number is an implicit packet sequence number represented as uint32. The sequence number is initialized to zero for the first packet, and is incremented after every packet, regardless of whether encryption or MAC is in use. It is never reset, even if keys or algorithms are renegotiated later. It wraps around to zero after every 2^32 packets. The packet sequence number itself is not included in the packet sent over the wire.

The MAC algorithms for each direction must run independently, and implementations must allow choosing the algorithm independently for both directions. The MAC bytes resulting from the MAC algorithm must be transmitted without encryption as the last part of the packet. The number of MAC bytes depends on the algorithm chosen. The default MAC algorithm defined is the hmac-sha1 (with digest length = key length = 20).

- Certificate format: The default certificate format used is ssh-dss signed with Simple DSS. Signing and verifying using this key format is done according to the Digital Signature Standard [FIPS-186] using the SHA-1 hash. A description can also be found in [SCHNEIR].
- Key exchange protocol: The default key exchange protocol is diffie-hellman-group1-sha1. The diffie-hellman-group1- sha1 method specifies Diffie-Hellman key exchange with SHA-1 as HASH.

Sections 5.12.2.1 and 5.12.2.2 briefly describe the implementation of SSH client and SSH server. For detailed information about the SSH Transport Layer Protocol, SSH Authentication Protocol, SSH Connection Protocol, and SSH Protocol Architecture, refer to the corresponding protocol documents at http://www.ietf.org/ids.by.wg/secsh.html.

5.12.2.1 SSH client

The ssh client first parses arguments and reads the configuration (`readconf.c`), then calls `ssh_connect()` (in `sshconnect*.c`) to open a connection to the server, and performs authentication (`ssh_login()` in `sshconnect.c`). Terminal echo is turned off while users type their passwords. SSH prevents the password from being displayed on the terminal as it is being typed. The SSH client then makes requests such as allocating a pseudo-tty, forwarding X11 connections, forwarding TCP-IP connections and so on, and might call code in `ttymodes.c` to encode current tty modes. Finally, the SSH client calls `client_loop()` in `clientloop.c`.

The client is typically installed suid root. The client temporarily gives up this right while reading the configuration data. The root privileges are used to make the connection from a privileged socket, which is required for host-based authentication and to read the host key for host-based authentication using protocol version 1. Any extra privileges are dropped before calling `ssh_login()`. Because `.rhosts` support is not included in the TSF, SSH the client is not suid root on the system.

5.12.2.2 SSH server daemon

The `sshd` daemon starts by processing arguments and reading the `/etc/ssh/sshd_config` configuration file. The configuration file contains keyword-argument pairs, one per line. Refer to the `sshd_config` man page for available configuration options. It then reads the host key, starts listening for connections, and generates the server key. An alarm regenerates the server key every hour.

When the server receives a connection, it forks a process, disables the regeneration alarm, and starts communicating with the client. The server and client first perform identification string exchange, and then negotiate encryption and perform authentication. If authentication is successful, the forked process sets the effective user ID to that of the authenticated user, performs preparatory operations, and enters the normal session mode by calling `server_loop()` in `serverloop.c`.

5.12.3 Very Secure File Transfer Protocol daemon

Very Secure File Transfer Protocol daemon (VSFTPD) provides a secure, fast, and stable file transfer service to and from a remote host. The behavior of VSFTPD can be controlled by its configuration file `/etc/vsftpd/vsftpd.conf`. The remainder of this section describes some of the security-relevant features of VSFTPD. For additional information, on SLES systems see the documents in `/usr/share/doc/packages/vsftpd/SECURITY/*`, and also http://vsftpd.beasts.org.

VSFTPD provides the following security-relevant features:

- Ability to use PAM to perform authentication.
- Ability to disable anonymous logins. If enabled, prevents anonymous users from writing.
- Ability to lock certain users in chroot jail in their home directories.
- Ability to hide all user and group information in the directory listing.
- Ability to set the secure tunneling scheme.
- Ability to perform enhanced logging.
- Ability to set connection timeout values.

The daemon generally follows these steps:

1. Parses command-line arguments.
2. Parses the configuration file.
3. Performs sanity checks such as ensuring that standard input is a socket.
4. Initializes the session.
5. Sets up the environment.
6. Starts logging.
7. Depending on the configuration, starts one or multiple process sessions.
8. Invokes appropriate functions to initiate connections.
9. Invokes `handle_local_login()` for non-anonymous users.
10. `handle_local_login()` invokes `vsf_sysdep_check_auth()` to perform authentication.
11. Performs authentication by PAM and starts the session. PAM does the following:
    1. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
    2. Invokes `pam_authenticate()` to authenticate the application user. Terminal echo is turned off while users are typing their passwords.
    3. Invokes `pam_acct_mgmt()` to perform module specific account management.
    4. Invokes `pam_setcred()` to set credentials.
    5. Invokes `pam_end()`.

5.12.4 CUPS

CUPS, the Common UNIX Printing System, is a portable printing layer for operating systems based on UNIX. It provides command-line interfaces for System V and BSD.

For background on CUPS labeled printing, please see: http://free.linux.hp.com/~mra/docs/.

CUPS uses the Internet Printing Protocol (IPP) that was designed to replace the Line Printer Daemon (LPD) protocol, as a basis for managing print jobs. CUPS also supports LPD, Server Message Block (SMB), and AppSocket protocols with reduced functionality. CUPS controls access to printers via its configuration file. For an overview of CUPS, refer to http://www.cups.org/documentation.php/overview.html or http://en.wikipedia.org/wiki/Common_Unix_Printing_System. For further information, refer to http://www.tldp.org/HOWTO/Printing-HOWTO/index.html or http://www.cups.org web sites.

5.12.4.1 cupsd

The `cupsd` daemon is the CUPS print server. It accepts local and remote print jobs, performs transformations on the data, and sends the final data to a printer. Daemon management is restricted to administrative users. Jobs are accepted only if they pass both CUPS access checks.

The `cupsd` daemon typically follows these processing steps:

1. Processes command line arguments.
2. Sets up signal handlers.
3. Forks a child.
4. The parent waits for the child to complete initialization (SIGUSR1) or die trying (SIGCHLD) and exits.
5. The child changes its working directory to /.
6. Disables core dumps.
7. Disconnects from the controlling terminal by running in a new session.
8. Closes all files.
9. Gets an audit file descriptor.
10. Sets the timezone.
11. Sets the locale.
12. Sets the resource limit for the maximum number of files.
13. Allocates memory for `select()` file descriptor sets.
14. Reads its configuration file.
15. Starts the server.
16. Sets up signal handlers.
17. Initializes authentication certificates.
18. Sends a `SIGUSR1` to the parent process so the parent can exit.
19. Starts any pending jobs.
20. Reaps dead children.
21. Closes idle clients.
22. Checks for new jobs.
23. Restarts the server if there are no active clients or jobs, or the reload timeout has been reached – this causes `cupd` to stop the server, reread the configuration file, and restart the server.

24. Check for input or output requests with `select()`.
25. If `select()` fails, logs error messages, notifies clients, and exits the main loop for shutdown processing.
26. Gets the current time.
27. Checks print status of print jobs.
28. Updates CGI data.
29. Updates notifier messages.
30. Expires subscriptions and removes completed jobs.
31. Updates the browse list.
32. Checks for new incoming connections on listening sockets and accepts them.
33. Checks for new data on client sockets.
34. Processes the client input.
35. Writes data back to the clients.
36. Closes inactive clients.
37. Updates pending multi-file documents.
38. Updates the root certificate every time a 5 minute timer has elapsed.
39. Goes back to step 20.
40. Upon exit from the main loop:
41. Logs a status message.
42. Stops the server.
43. Frees all jobs.
44. Frees file descriptor sets.
45. Closes audit file descriptor.
46. Exits.

5.12.4.2 ping

`ping` opens a raw socket and uses the ICMP protocol's mandatory `ECHO_REQUEST` datagram to elicit an `ICMP ECHO_RESPONSE` from a host or a gateway. An `ECHO_REQUEST` datagram, or ping, has an IP and ICMP header, followed by a `struct timeval`, and then an arbitrary number of pad bytes used to fill out the packet. For more information on the ping command, see the ping(8) man page.

5.12.4.3 ping6

The `ping6` is the IPv6 counterpart to the IPv4 ping command (q.v.). It sends ICMPV6 ECHO_REQUESTs to elicit ECHO_RESPONSEs. Use of ping6 requires the `CAP_NET_RAWIO` capability. For more information on the ping6 command, see the ping6(8) man page.

5.12.4.4 openssl

`openssl` is a command-line interface to the OpenSSL cryptography toolkit, which implements the Secure Socket Layer (SSL v2j and v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards that they require. The `openssl` command can be used by an administrative user for the following:

- Creation of RSA, DH, and DSA parameters.
- Generation of 1024-bit RSA keys.
- Creation of X.509 certificates, CSRs, and CRLs.
- Calculation of message digests.
- Encryption and Decryption with ciphers.
- SSL and TLS client and server tests.
- Handling of S/MIME signed or encrypted mail.

For detailed information about the openssl command and its usage, see: http://www.openssl.org/docs/apps/openssl.html.

5.12.4.5 stunnel

`stunnel` is designed to work as an SSL encryption wrapper between remote clients and local or remote servers. `stunnel` can be used to add SSL functionality to commonly used daemons such as POP and IMAP servers, to standalone daemons like SMTP and HTTP, and in tunneling PPP over network sockets without changes to the source code.

The most common use of `stunnel` is to listen on a network port and establish communications with either a new port via the connect option, or a new program via the exec option. There is also an option that allows a program to accept incoming connections and then launch `stunnel`.

Each SSL-enabled daemon needs to present a valid X.509 certificate to the peer. The SSL-enabled daemon also needs a private key to decrypt incoming data. `stunnel` is built on top of SSL, so on the TOE the private key and the certificate can be generated by OpenSSL utilities. These private keys are stored in the `/etc/stunnel/stunnel.pem` file.

`stunnel` uses the openssl library, and therefore can use the cipher suites implemented by that library. They are:

- SSL_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_
- SSL_RSA_WITH_3DES_EDE_CBC_SHA

`stunnel` is configured by the `/etc/stunnel/stunnel.conf` file. The file is a simple ASCII file that can be edited by the administrative user to secure SSL-unaware servers. Each service to be secured is named in a square bracket, followed by "option_name = option_value" pairs for that service. Global parameters such as location of the private key file are listed at the beginning of the file. An example follows:

```
# Global parameters
cert = /etc/stunnel/stunnel.pem
pid = /tmp/stunnel.pid
setuid = nobody
setgid = nogroup
```

```
# Service-level configuration
# --------------------------

[ssmtp]
accept = 465
connect = 25
```

The above configuration secures localhost-SMTP when someone connects to it via port 465. The configuration tells `stunnel` to listen to the SSH port 465, and to send all info to the plain port 25 on localhost.

For additional information about `stunnel`, refer to its man page as well as http://stunnel.mirt.net and http://www.stunnel.org.

5.12.4.6 xinetd

The `xinetd` daemon dispatches children to service incoming requests. For more information on `xinetd`, see the SLES Security Guide or the xinetd(8) man page.

5.13 System management

5.13.1 Account Management

5.13.1.1 chage

The `chage` program allows a system administrator to alter a user's password expiration data. See the `chage` man page for more information. `chage` generally follows these steps.

1. Sets language.
2. Sets up a variable indicating whether the application user is the root user.
3. Parses command-line arguments.
4. Performs a sanity check on command-line arguments.
5. If the application user is not root, allows only the listing of the user's own password age parameters.
6. Invokes getpwuid (`getuid()`) to obtain the application user's passwd structure.
7. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
8. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
9. Invokes `pam_acct_mgmt()` to perform module specific account management.
10. If called to list password age parameters, lists them now and exits.
11. Locks and opens authentication database files.
12. Updates appropriate database files with new password age parameters.
13. Closes database files.

14. Invokes `pam_chauthok()` to rejuvenate user's authentication tokens.
15. Exits.

5.13.1.2 chfn

The `chfn` program allows users to change their finger information. The `finger` command displays the information, stored in the `/etc/passwd` file. Refer to the `chfn` man page for detailed information. `chfn` generally follows these steps:

1. Sets language.
2. Gets invoking user's ID.
3. Parses command-line arguments.
4. Performs a check that a non-root user is not trying to change finger information of another user.
5. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
6. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
7. Invokes `pam_acct_mgmt()` to perform module-specific account management.
8. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
9. Invokes `pam_setcred()` to set credentials.
10. Prompts for new finger information if not supplied on the command line.
11. Updates appropriate database files with new finger information.
12. Exits.

5.13.1.3 chsh

The `chsh` program allows users to change their login shells. If a shell is not given on the command line, `chsh` prompts for one. Refer to the `chsh` man page for detailed information. `chsh` generally follows these steps:

1. Sets language.
2. Gets invoking user's ID.
3. Parses command-line arguments.
4. Performs a check that a non-root user is not trying to change shell of another user.
5. Performs a check to ensure that a non-root user is not trying to set his or her shell to a non standard shell.
6. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
7. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
8. Invokes `pam_acct_mgmt()` to perform module-specific account management.
9. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
10. Checks the shell to make sure that it is accessible.

11. Invokes `setpwnam()` to update appropriate database files with the new shell.
12. Exits.

5.13.2 User management

5.13.2.1 useradd

The `useradd` program allows an authorized user to create new user accounts on the system. Refer to the `useradd` man page for more information. `useradd` generally follows these steps:

1. Sets language.
2. Invokes getpwuid (`getuid()`) to obtain the application user's passwd structure.
3. Invokes `pam_start()` to initialize the PAM library, and to identify the application with a particular service name.
4. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
5. Invokes `pam_acct_mgmt()` to perform module-specific account management.
6. Gets the default parameters for a new user account from `/etc/default/useradd`.
7. Processes command-line arguments.
8. Ensures that the user account being created doesn't already exist.
9. Invokes `open_files()` to lock and open authentication database files.
10. Invokes `usr_update()` to update authentication database files with new account information.
11. Generates audit records to log actions of the `useradd` command. Actions such as addition of new user, addition of user to a group, update of default user parameters, and creation of a user's home directory.
12. Invokes `close_files()` to close authentication database files.
13. Creates a home directory for the new user.
14. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
15. Exits.

5.13.2.2 usermod

The `usermod` allows an administrator to modify an existing user account. Refer to the `usermod` man page for more detailed information on the usage of the command. `usermod` generally follows these steps:

1. Sets language.
2. Invokes getpwuid (`getuid()`) to obtain application user's passwd structure.
3. Invokes `pam_start()` to initialize the PAM library, and to identify the application with a particular service name.
4. Invokes `pam_authenticate()` to authenticate the application user. Generates audit record to log the authentication attempt and its outcome.
5. Invokes `pam_acct_mgmt()` to perform module-specific account management.

6. Processes command-line arguments.
7. Ensures that the user account being modified exists.
8. Invokes `open_files()` to lock and open authentication database files.
9. Invokes `usr_update()` to update authentication database files with updated account information.
10. Generates audit record to log actions of the `usermod` command. The logged actions include locking and unlocking of user account, changing of user password, user name, user ID, default user group, user shell, user home directory, user comment, inactive days, expiration days, mail file owner, and moving of user's home directory.
11. If updating group information, invokes `grp_update()` to update group information.
12. Invokes `close_files()` to close authentication database files.
13. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
14. Exits.

5.13.2.3 userdel

The `userdel` program allows an administrator to delete an existing user account. Refer to the `userdel` man page for more information. `userdel` generally follows these steps:

1. Sets language.
2. Invokes getpwuid (`getuid()`) to obtain the application user's passwd structure.
3. Invokes `pam_start()` to initialize PAM library and to identify the application with a particular service name.
4. Invokes `pam_authenticate()` to authenticate the application user. Generates audit record to log the authentication attempt and its outcome.
5. Invokes `pam_acct_mgmt()` to perform module-specific account management.
6. Processes command-line arguments.
7. Ensures that the user being deleted does exist, and is currently not logged on.
8. Invokes `open_files()` to lock and open authentication database files.
9. Invokes `usr_update()` to update authentication database files with updated account information.
10. Invokes `grp_update()` to update group information.
11. Generates audit record to log deletion of a user and the deletion of user's mail file.
12. Invokes `close_files()` to close authentication database files.
13. If called with the `-r` flag, removes the user's mailbox by invoking `remove_mailbox()` and removes the user's home directory tree by invoking `remove_tree()`.
14. Cancels any `cron` or `at` jobs that the user created.
15. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
16. Exits.

5.13.3 Group management

5.13.3.1 groupadd

The `groupadd` program allows an administrator to create new groups on the system. Refer to the `groupadd` man page for more detailed information on usage of the command. `groupadd` generally follows these steps:

1. Sets language.
2. Invokes getpwuid (`getuid()`) to obtain an application user's passwd structure.
3. Invokes `pam_start()` to initialize the PAM library, and to identify the application with a particular service name.
4. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
5. Invokes `pam_acct_mgmt()` to perform module-specific account management.
6. Processes command-line arguments.
7. Ensures that the group being created does not already exist.
8. Invokes `open_files()` to lock and open authentication database files.
9. Invokes `grp_update()` to update authentication database files with new group information. Generates audit record to log creation of new group.
10. Invokes `close_files()` to close the authentication database files.
11. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
12. Exits.

5.13.3.2 groupmod

The `groupmod` program allows an administrator to modify existing groups on the system. Refer to the `groupmod` man page for more information. `groupmod` generally follows these steps:

1. Sets language.
2. Invokes getpwuid (`getuid()`) to obtain application user's passwd structure.
3. Invokes `pam_start()` to initialize the PAM library, and to identify the application with a particular service name.
4. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
5. Invokes `pam_acct_mgmt()` to perform module-specific account management.
6. Processes command-line arguments.
7. Ensures that the group being modified does exist.
8. Invokes `open_files()` to lock and open authentication database files.
9. Invokes `grp_update()` to update authentication database files with updated group information. Generates audit record to log updates to existing groups.
10. Invokes `close_files()` to close authentication database files.
11. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
12. Exits.

5.13.3.3 groupdel

The `groupdel` program allows an administrator to delete existing groups on the system. Refer to the `groupdel` man page for more information. `groupdel` generally follows these steps:

1. Sets language.

   Invokes getpwuid (`getuid()`) to obtain the application user's passwd structure.
2. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
3. Invokes `pam_authenticate()` to authenticate the application user. Generates an audit record to log the authentication attempt and its outcome.
4. Invokes `pam_acct_mgmt()` to perform module-specific account management.
5. Processes command-line arguments.
6. Ensures that the group being deleted does exist, and that it is not the primary group for any users.
7. Invokes `open_files()` to lock and open authentication database files.
8. Invokes `grp_update()` to update group information. Generates an audit record to log deletion of existing groups.
9. Invokes `close_files()` to close the authentication database files.
10. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
11. Exits.

5.13.4 System Time management

5.13.4.1 date

The `date` program, for a normal user, displays current date and time. For an administrative user, `date` can also set the system date and time. Refer to the `date` man page for more information. `date` generally follows these steps:

1. Sets language.
2. Parses command-line arguments.
3. Validates command-line arguments.
4. If command line options indicate a system time set operation, invokes the `stime()` system call to set the system time. The system call handler routine for `stime()` checks if the process possesses the `CAP_SYS_TIME` capability. If it does, the operation is allowed; otherwise, an error is returned.
5. Processes returns from the `stime()` system call. Prints current time or error, depending on the return value from the system call.
6. Exits.

5.13.4.2 hwclock

The `hwclock` command displays the current hardware clock time for a normal user. For an administrative user, `hwclock` can also set the hardware clock time. It also allows the administrative user to set system time from the hardware clock time. The `hwclock` man page gives more information. `hwclock` follows these steps:

1. Sets language
2. Parses command-line arguments
3. Validates command-line arguments
4. If command-line options indicate a hardware clock set operation, then it opens the `/dev/rtc` device special file, and performs I/O to set the value and kernel ensures that process has `CAP_SYS_TIME` capability.
5. Or, if a user mentions uses the `–directisa` option, then `hwclock` uses explicit I/O instructions to access CMOS memory registers, which sets the clock value. Here also the kernel ensures that process has `CAP_SYS_TIME` capability.
6. If `hwclock` succeeds, it sets the value of hardware clock; otherwise, it gives the appropriate error message.
7. Exits.

5.13.5 Other System Management

5.13.5.1 AMTU

Abstract machine test utility (AMTU): The TOE security functions are implemented using underlying hardware. The TSF depends on the hardware to provide certain functionalities in order for the security functions to work properly. Because the TOE includes different hardware architectures, a special tool is provided to test features of the underlying hardware that the TSF depends on.

This tool works from a premise that it is working on an abstract machine that is providing functionality to the TSF. The test tool runs on all hardware architectures that are targets of evaluation and reports problems with any underlying functionalities. For more detailed information on the Abstract Machine Test, refer to Emily Ratliff, "Abstract Machine Testing: Requirements and Design." The AMTU test tool performs detailed in the subsections that follow.

5.13.5.1.1 Memory

AMTU allocates 10% of the physical memory of the system (not exceeding 1 GB on 32 bit machines), and then writes a pattern of random bytes. The tool reads back the memory and ensures that what was read matches what was written. If they do not match, the tool reports a memory failure.

5.13.5.1.2 Memory separation

To fulfill the memory separation requirement, AMTU performs the following: As a normal user, the tool picks random areas of memory in ranges reported in `/proc/ksyms` to ensure that user-space programs cannot read from, and write to, areas of memory utilized by such things as Video RAM and kernel code. The tool reports a failure if any of the above attempts succeed.

5.13.5.1.3 I/O controller and network

Because portions of the TSF depend on the reliability of the network devices and the disk controllers, AMTU also checks I/O devices. This section describes how the network devices are tested.

When the kernel detects an attempt to open a network connection to an address that is configured on the local machine, the kernel short-circuits the packets rather than sending them to the physical device. To evade this optimization without requiring a remote server, the tool specifies the `PF_PACKET` communication domain (see packet (7)) when opening the socket. AMTU performs the following:

1. Using the `PF_PACKET` communication domain, opens another connection to the listening server and
2. Ensures that the random data transmitted is also the data received. Steps 1 and 2 are repeated for each configured network device.

5.13.5.1.4 I/O controller and disk

In order to check the disk controllers (IDE and SCSI only), AMTU opens a file on each read-write mounted file-system, writes a 10 MB random string, syncs the file and directory, closes the file, re-opens the file, and reads it to validate that the string is unchanged. The 10 MB string size is chosen so that the file exceeds the size of the device's buffer. AMTU prints a warning to the administrator if it has determined that a disk controller (IDE only) was not tested, unless that disk controller is dedicated to floppy and cdrom devices. This might happen if a disk controller only controls read-only file systems. More than one test is performed on disk controllers that control more than one r/w file system.

5.13.5.1.5 Supervisor mode instructions

Certain instructions are only available in supervisor mode. The kernel has the ability to switch to supervisor mode to use the instructions, but user space tools should not be able to use these instructions. A subset of these privileged instructions are tested to confirm that is true.

The list of instructions that are available only in supervisor mode is architecture dependent. The subset of the privileged instructions that are tested per platform is listed below. In addition, to generically test that privileged instructions cannot be executed while not in supervisor mode, the test ensures that the CPU control registers, task registers, and interrupt descriptor tables cannot be changed while not in supervisor mode. Supervisor instructions available for each of the architecture are described in subsections below.

5.13.5.1.5.1 System p

The instruction set for the PowerPC processor is given in the book at the following URL: http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600682CC7/$file/booke_rm.pdf

For each instruction, the description in the book lists whether it is available only in supervisor mode or not. The following instructions are tested by the AMTU:

- `TLBSYNC`: TLB Synchronize
- `MFSR`: Move from Segment Register
- `MFMSR`: Move From Machine State Register

The expected outcome from attempting to execute these instructions is an ILLEGAL Instruction signal (`SIGILL` – 4).

5.13.5.1.5.2 System z

The book entitled *Principles of Operation* is a handy reference for the System z architecture:

http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9AR006/CCONTENTS

The following privileged instructions are tested by this tool:

- `PTLB`: Purge TLB
- `RRBE`: Reset reference bit extended
- `PALB`: Purge ALB
- `EPAR`: Extract Primary ASN
- `HSCH`: Halt subchannel
- `LPSW`: Load PSW (To test the CPU control register).

The expected outcome from attempting to execute these instructions is an ILLEGAL Instruction signal (`SIGILL` – 4).

5.13.5.1.5.3 System x

Section 4.9 from the *Intel Architecture Software Developer's Manual Volume 3: System Programming* book at ftp://download.intel.com/design/PentiumII/manuals/24319202.pdf gives a list of privileged instructions available for the x86 architecture.

This tool tests the following privileged instructions:

- `HLT`: halt the processor
- `RDPMC`: read performance-monitoring counter
- `CLTS`: Clear task-switched flag in register CR0.
- `LIDT`: Load Interrupt Descriptor Table Register
- `LGDT`: Load Global Descriptor Table Register
- `LTR`: Load Task Register
- `LLDT`: Load Local Descriptor Table Register

To test CPU control registers, use `MOVL %cs, 28(%esp)`. This overwrites the value of the register that contains the code segment. The register that contains the address of the next instruction (`eip`) is not directly addressable. Note that in the Intel documentation of `MOV` it is explicitly stated that `MOV` cannot be used to set the `CS` register. Attempting to do so will cause an exception (`SIGILL` rather than `SIGSEGV`).

The expected outcome of attempting to execute these instructions is a segmentation violation signal (`SIGSEGV` – 11).

5.13.5.1.5.4 eServer 326

Chapter 4 of the *AMD Architecture Programmer's Manual Volume 3: General Purpose and System* instructions at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf gives the list of privileged instructions available for the AMD64 architecture.

This tool tests the following privileged instructions:

- `HLT`: Halt the processor
- `RDPMC`: Read performance-monitoring counter
- `CLTS`: Clear task-switched flag in register CR0.
- `LIDT`: Load Interrupt Descriptor Table Register
- `LGDT`: Load Global Descriptor Table Register
- `LTR`: Load Task Register
- `LLDT`: Load Local Descriptor Table Register

To test CPU control registers, use `MOVL %cs, 28(%esp)`. This overwrites the value of the register that contains the code segment. The register that contains the address of the next instruction (`eip`) is not directly addressable. `MOV` cannot be used to set the `CS` register. Attempting to do so will cause an exception (`SIGILL` rather than `SIGSEGV`).

The expected outcome of attempting to execute these instructions is a Segmentation Violation signal (`SIGSEGV` – 11).

5.13.5.1.6 AMTU output

For each of these subsystems, the AMTU tool reports what aspect of the system it is currently testing and then reports either success or failure. This message is also logged to the audit subsystem. In the case of failure, any additional information available is reported to the system administrator to help troubleshoot the problem.

5.13.5.2 star

The `star` utility is an extended version of the tape archive command. It can save and restore the extended attributes the ext3 filesystem uses to store ACLs. For more information on the star utility, see the star(1) man page.

The star utility typically follows these processing steps:

1. Saves the command line arguments.

2. Gets its euid and uid.
3. Transforms old-style command line argument syntax into new-style syntax.
4. Processes the command line arguments.
5. Sets up signal handling.
6. Initializes the fifo.
7. Initializes any remote connection.
8. Sets back the real UID.
9. Opens the archive.
10. Initializes data structures.
11. Checks for multi-volume format; logs an error and exits if so because it is unsupported.
12. Initializes device macro handling.
13. Initializes extended headers buffer.
14. Initializes UID and GID handling.
15. Gets the working directory.
16. Gets the current time.
17. Initializes the date name.
18. Initializes the volume header.
19. Checks for a FIFO; forks and runs operation if so.
20. Checks for a copy flag; copies the archive if so.
21. Checks for TOC, create, or extract flags.
22. If a list file arguments was passed, opens that and uses it as the list of files to process.
23. If no file list was given, does a find for each of the command line arguments.
24. Closes the pattern.
25. If the TOC flag was given, lists the files.
26. If doing an extract, changes to the change dir argument, if present, and extracts the files,
27. If doing a diff, shows the file differences.
28. Closes the pattern.
29. Performs tape manipulation.
30. If it is a create, creates the archive.
31. Checks archive links.
32. Closes the archive.
33. Prints any statistics requested.
34. Checks for errors and prints error codes.
35. Flushes `stdout` if on a terminal.
36. Writes out dump dates if necessary.
37. Exits.

5.13.6 I&A support

5.13.6.1 pam_tally

The `pam_tally` utility allows administrative users to reset the failed login counter kept in the `/var/log/faillog`. Please see the `/usr/share/doc/packages/pam/modules/README.pam_tally` file on a SLES system for more information.

5.13.6.2 unix_chkpwd

The `unix_chkpwd` helper program works with the `pam_unix` PAM module (Section 5.11.1.3). It is intended only to be executed by the `pam_unix` PAM module and logs an error if executed otherwise. For more information on the `unix_chkpwd` helper program, please see the unix_chkpwd(8) man page.

The `unix_chkpwd` helper program typically follows these processing steps:

1. Sets up a signal handler.
2. Checks that it is not running on a TTY.
3. Gets the user's name.
4. Verifies the password if passed the verify command line argument.
5. Updates the shadow file if passed the update command line argument.
6. Reads the password from `stdin`.
7. Validates the length of the password.
8. Verifies the password against the shadow database.
9. Zeros the password memory.
10. Exits.

5.14 Batch processing

Batch processing on the SLES system means to submit a job that will be run when the system load permits. Batch processing allows users to perform CPU-intensive tasks while the system load is low; it also allows users and system administrators to automate routine maintenance tasks. While batch processing provides a convenient feature, it also raises a security issue, because a privileged process has to perform a task ordered by a normal user.

This section describes different trusted commands and processes that implement the batch processing feature. Mechanisms are highlighted that ensure how normal users are prevented from performing actions for which they are not authorized. Batch processing is implemented with the `crontab`, `batch` and `at` user commands, and the `cron` and `atd` trusted processes. The command `batch` is a script that invokes at; therefore, only `at` internals are described in this section.

5.14.1 Batch processing user commands

5.14.1.1 crontab

`crontab` is the batch processing user command. `crontab` uses a control file to dictate when repeated jobs will execute.

The crontab program is used to install, deinstall, or list the tables used to drive the cron daemon in Vixie Cron. The crontab program allows an administrator to perform specific tasks on a regularly-scheduled basis without logging in. Users can have their own crontabs that allow them to create jobs that will run at given times. A user can create a crontab file with the help of this command. The crontab command generally goes through these steps:

1. Parses command-line options to determine if the crontab file is to be created, listed, edited or replaced.
2. Checks if the user is authorized to use this command. If the /etc/cron.allow file exists, only users listed in that file are allowed to use this command. If the /etc/cron.deny file exists, then users listed in that file are not allowed to use this command. It generates an audit record if a user is not allowed to use it.
3. If listing, crontab invokes the list_cmd() routine to list the existing crontab file. It generates an audit record to log the listing of crontab files.
4. If deleting, crontab invokes the delete_cmd() routine to delete the existing crontab file. It generates an audit record to log the deletion of an existing crontab file.
5. If editing a crontab, it invokes the edit_cmd() routine to edit the existing crontab file. It generates audit record to log modification of an existing crontab file.
6. If replacing a crontab, crontab invokes the replace_cmd() routine to replace the existing crontab file. After the edit and replace option, crontab ensures that the modified new crontab file is owned by root and has an access mode of 600. It generates an audit record to log the replacement of an existing crontab file.

crontab files are created in the /var/spool/cron/ directory and are created with the login name of the respective user. This establishes the identity of the user on whose behalf commands will be executed. Since the /var/spool/cron directory is owned by root and has an access mode of 700, normal users cannot schedule jobs in the name of other users.

5.14.1.2 at

The at command executes commands at a specified time and optional date. The commands are read from standard input or from a file. at is also used for performing maintenance, such as listing and removing existing jobs. at generally follows these steps:

1. Registers if it was called as at, atq or atrm, to create at jobs, list at jobs, or remove at jobs, respectively.
2. Checks to ensure that the user is allowed to use this command. at command can always be issued by a privileged user. Other users must be listed in the file /etc/at.allow if it exists; otherwise, they must not be listed in /etc/at.deny. If neither file exists, only a privileged user can issue the command. If a user is not allowed to use this command to create an at job, generates an audit record to log the attempt.
3. If called as atq, invokes list_jobs() to list existing at jobs. atq changes directory to /var/spool/atjobs, reads its directory content, and lists all existing jobs queued for execution.
4. If called as atrm, invokes process_jobs() to remove existing jobs. atrm changes directory to /var/spool/atjobs and unlinks the appropriate job file.
5. If called as at, parses the time argument and calls writefile() to create a job file in /var/spool/atjobs. Generates an audit record to log the creation of an at job. The job file is owned by the invoking user and contains current umask and environment variables along with the

commands that are to be executed. Information stored in this job file, along with its attributes, is used by the `atd` daemon to recreate the invocation of the user's identity while performing tasks at the scheduled time.

5.14.2 Batch processing daemons

5.14.2.1 cron

The `cron` daemon executes commands scheduled through `crontab` or listed in `/etc/crontab` for standard system `cron` jobs.

The `cron` trusted process daemon processes users' `crontab` files. The `cron` daemon ensures that the system DAC policy is not violated by duplicating the login environment of the user whose `crontab` file is being processed. The `cron` daemon depends on the `crontab` trusted command to create the `crontab` file of each user with his or her name. The `/var/spool/cron/tabs/root` file contains the `crontab` for root, and therefore is critical. The `cron` daemon also depends on the kernel's file system subsystem to prevent normal users from creating or modifying other users' `crontab` files. The `cron` daemon starts during system initialization, and generally follows these steps:

1. Sits in an infinite loop, waking up after one minute to process `crontab` files.
2. Sets the system's `cron` jobs by reading `crontab` files in the directory `/etc/cron.d/`.
3. Sets `cron` jobs to be executed weekly, hourly, daily and monthly by reading their respective `crontab` files from directories `/etc/cron` {weekly hourly daily monthly}.
4. Calls the `load_database()` routine to read `crontab` files in the `/var/spool/cron/tabs` directory.
5. For every `crontab` file, invokes `getpwnam()` to get the user's identity information.
6. For each `crontab` file, at the appropriate time, which is set in the file, the daemon forks a child to execute commands listed in the `crontab` file. The child sets its credentials based on the user's login environment before executing any commands. It generates audit records to log execution of `cron` jobs.

5.14.2.2 atd

The `atd` is the trusted process daemon that services users' requests for timed execution of specific tasks. The `atd` ensures that the system's DAC policy is not violated by exactly duplicating the identity for the user on whose behalf it is performing tasks. The `atd` depends on the trusted command `at` to have appropriately created at jobs file containing pertinent information about the user's identity. The `atd` is started during system initialization time and generally goes through these steps:

1. Attaches to the audit subsystem.
2. On a regular interval or on receiving a signal from a user looks into the `/var/spool/atjobs` directory for processing jobs.
3. If an appropriate job is found, forks a child process and sets its user and group IDs to those of the owner of the job file. Sets up standard out to go to a file. Performs the tasks listed in the job file by executing the user's shell and e-mails the user when the job is finished. Generates audit record to log processing of an at job.

5.15 User-level audit subsystem

The main user-level audit components consist of the auditd daemon, the auditctl control program, the libaudit library, the auditd.conf configuration file, and the auditd.rules initial setup file. There is also the /etc/init.d/auditd init script that is used to start and stop auditd. When run, this script sources another file, /etc/sysconfig/auditd, to set the locale, and to set the AUDIT_CLEAN_STOP variable, which controls whether to delete the watch points and the filter rules when auditd stops.

On startup, auditd reads the configuration file to set the various configuration options that pertain to the daemon. Then, it reads the auditd.rules file to set the initial rules. The auditd.conf man page describes all the configurable options. The auditctl man page lists all the supported control options.

5.15.1 Audit daemon

The auditd daemon does the following on startup:

1. Registers its pid with the kernel, so the kernel starts sending all audit events to the daemon (to the netlink).
2. Enables auditing.
3. Opens the netlink socket, and spawns a thread that continuously waits on the condition of audit record data availability on the netlink. Once the data is available it signals the thread, which writes out the audit records.
4. Reads the /etc/auditd.conf configuration file, which holds the configuration parameters that define, among other things, what to do when errors are encountered or when the log files are full.
5. Usually, the /etc/init.d/auditd init script runs auditd, which issues auditctl -R /etc/audit.rules, if /etc/auditd.rules exists.
6. auditctl can be used at any time, even before auditd is running, to add and build rules associated with possible actions for system calls and file system operations. It also sets the behavior of the audit subsystem in the kernel.
7. If audit is enabled, the kernel intercepts the system calls and generates audit records according to the filter rules. Or, it generates audit records for watches set on particular file system files or directories.
8. Trusted programs can also write audit records for security-relevant operations through the audit netlink, and not directly to the audit log.

5.15.2 Audit utilities

In addition to the main components, the user level provides the ausearch search utility and the autrace trace utility. While ausearch finds audit records based on different criteria from the audit log, autrace audits all syscalls issued by the process being traced. The man pages for these two utilities detail all the options that can be used. This section only describes how they operate.

5.15.2.1 aureport

The aureport utility provides summary information from audit log files. Use of aureport is restricted to administrative users. For more information on the aureport utility, see the aureport(8) man page.

aureport typically follows these processing steps:

1. Sets the locale.

2. Processes the command line arguments.
3. Attempts to raise its resource limits.
4. Sets its umask.
5. Resets its internal counters.
6. Emits a title.
7. Processes audit records from an audit log file or `stdin`, incrementing counters depending on audit record contents.
8. Prints a message and exits if there are no useful events.
9. Prints a summary report.
10. Destroys its data structures and frees memory.
11. Exits.

5.15.2.2 ausearch

Only root has the ability to run this tool. First, `ausearch` checks the validity of the parameters passed, whether they are supported or not. Then, it opens either the logs or the administrator-specified files. The log's location is extracted from the `/etc/auditd.conf`. For more information on `ausearch`, see the ausearch(8) man page.

After that, `ausearch` starts to process the records one record at a time, matching on the parameters passed to it. Because each audit record can be written into the log as multiple file records, the tool collates all the file records into a linked list before it checks whether the record matches the requested search criteria.

5.15.2.3 autrace

Only root can run this command. `autrace` executes the program passed to it after setting a filter to audit all system calls for the new process. If any rules or watches were previously set, `autrace` will not run; it requires that all rules and watches be cleared first. For more information on `autrace`, see the autrace(8) man page.

5.15.3 Audit configuration files

See Section 5.6.2.1 Configuration for more detail on audit configuration files.

5.15.4 Audit logs

LAF audit logs, also known as audit trails, are the final repository of audit records generated by the kernel and the trusted programs. An administrative user can use the `ausearch` on audit logs to extract and analyze security-relevant events.

Audit logs are protected by their DAC mode, in order to protect them from unauthorized deletion or modification.

An administrator can specify in the `auditd.conf` file what actions `auditd` should perform whenever audit logs reach a specified size. Also, the administrator can specify what happens when writing to the audit logs encounters an error.

5.16 Supporting functions

Trusted programs and trusted processes in an SLES system use libraries. Libraries do not form a subsystem in the notation of the Common Criteria, but they provide supporting functions to trusted commands and processes.

A library is an archive of link-edited objects and their export files. A shared library is an archive of objects that has been bound as a module with imports and exports, and is marked as a shared object. When an object exported by a shared library is referenced, the loader checks for the object in the calling process's shared library segment.

If the library is there, the links are resolved, and the program can call the shared library code. If the library isn't there, the loader pages the library into the shared memory segment, where it can subsequently be used by other programs. This section briefly describes the library and system-call linking mechanism in user and kernel space, and illustrates any security implications.

5.16.1 TSF libraries

The following table lists some of the libraries that are used by trusted programs and processes. The libraries may also be used by untrusted programs, but are still part of the TSF. The libraries are protected from modification by the file system DAC mechanism.

Library	Description
/lib/libc.so.6	C Run time library functions.
/lib/libcrypt.so.1	Library that performs one-way encryption of user and group passwords.
/lib/libcrypt.so.o.9.8b	Replacement library for libcrypt.so.1 Supports bigcrypt and blowfish password encryption.
/lib/security/pam_unix.so	Modules that perform basic password-based authentication, configured with the MD5 hashing algorithm.
/lib/security/pam_passwdqc.so	Modules that enforce additional stricter password rules. For example, reject passwords that follow keyboard patterns such as “1az2pqr”.
/lib/security/pam_wheel.so	Modules that restrict use of the su command to members of the wheel group.
/lib/security/pam_nologin.so	Modules that allow the administrator to disable all logins with the /etc/nologin file.
/lib/security/pam_securetty.so	Modules that restrict root access to specific terminals.
/lib/security/pam_tally.so	Modules that deny access based on the number of failed login attempts.
/lib/security/pam_listfile.so	Modules that allow use of ACLs based on users, ttys, remote hosts, groups, and shells.
/lib/security/pam_deny.so	Module that always returns a failure.
/lib/security/pam_env.so	Module that loads a configurable list of environment variables.
/lib/security/pam_xauth.so	Module that forwards xauth cookies from user to user.
/lib/security/pam_limits.so	Module that imposes user limits on login.
/lib/security/pam_shells.so	Module that grants authentication if the users shell is listed in /etc/shells
/lib/security/pam_stack.so	Module that performs normal password authentication through recursive stacking of modules.
/lib/security/pam_rootok.so	An authentication module that performs one task: if the id of the user is '0' then it returns 'PAM_SUCCESS'
/usr/lib/libssl3.so	OpenSSL library with interfaces to Secure Socket Layer version 3 and Transport Layer Security version 1 protocols.
/lib/libcrypto.so.4	OpenSSL crypto library with interfaces to wide range of cryptographic algorithms used in various Internet standards.

Table 5-7: TSF libraries

5.16.2 Library linking mechanism

On SLES, a binary executable automatically causes the program loader `/lib/ld-linux.so.2` to be loaded and run. This loader takes care of analyzing the library names in the executable file, locating the library in the system directory tree, and making requested code available to the executing process. The loader does not copy the library object code, but instead performs a memory mapping of the appropriate object code into the address space of the executing process. This mapping allows the page frames containing the object code of the library to be shared among all processes that invoke that library function.

Page frames included in private regions can be shared among several processes with the Copy On Write mechanism. That is, the page frames can be shared as long as they are not modified. The page frames containing the library object code are mapped in the text segment of the linear address space of the program. Because the text segment is read-only, it is possible to share executable code from the library among all currently executing processes.

The kernel carries out this mapping of page frames in a read-only text segment, without any input from the user. Object code is shared in read-only mode, preventing one process from making an unauthorized modification to another process's execution context, thus satisfying the DAC requirement. Page frames used for this mapping are allocated with the demand paging technique, described in Section 5.5.2.5.6, which satisfies the object reuse requirement.

On SLES systems, the administrator can control the list of directories that are automatically searched during program startup. The directories searched are listed in the `/etc/ld.so.conf` file. A normal user is not allowed write access to the `/etc/ld.so.conf` file. The loader also allows certain functions to be overridden from shared libraries with environment variables `LD_PRELOAD` and `LD_LIBRARY_PATH`.

The `LD_PRELOAD` variable lists object files with functions that override the standard set. The `LD_LIBRARY_PATH` variable sets up lists of directories that are searched before loading from the standard directory list. In order to prevent a normal user from violating the security policy, these variables are ignored, and removed from the process's environment when the program being executed is either setuid or setgid.

The system determines if a program is setuid or setgid by checking the program's credentials; if the UID and EUID differ, or the GID and the EGID differ, then the system presumes the program is setuid or setgid, or descended from one, and does not allow preloading of user-supplied functions to override ones from the standard libraries. Similarly if the program execution results in domain transition then the system checks to see if the resulting domain possesses the type enforcement permission *noatsecure*. If noatsecure is not present then the system clears `LD_PRELOAD` and `LD_LIBRARY_PATH` variables and thus does not allow preloading of user-supplied functions to override ones from the standard libraries.

When an executable is created by linking with a static library, the object code from the library is copied into the executable. Because there is no sharing of page frames with other executing processes, there are no DAC or object reuse issues with static libraries.

5.16.3 System call linking mechanism

A system call is an explicit request to the kernel made via a software interrupt. The implementation of this interrupt is dependent on the hardware architecture. The following subsections briefly describe the system call interrupt setup for the different hardware architectures that are part of the TOE.

5.16.3.1 System x

On System x systems, the Interrupt Descriptor Table (IDT) for the Intel processors is initialized to allow a trap gate that can be accessed by a user-mode process. The `trap_init()` routine does this mapping at system initialization, and sets the IDT entry corresponding to vector 128 (Ox80) to invoke the system call exception handler.

When compiling and linking a program that makes a system call, the libc library wrapper routine for that system call stores the appropriate system call number in the `eax` register, and executes the int 0x80 assembly language instruction to generate the hardware exception. The exception handler in the kernel for this vector is the `system_call()` system call handler. `system_call()` saves the contents of registers in the kernel-mode stack, handles the call by invoking a corresponding C function in the kernel, and exits the handler by means of the `ret_from_sys_call()` function.

5.16.3.2 System p

On System p systems, the PowerPC and POWER architecture provides the assembly instruction supervisor call (SC) to make a system call. The kernel also uses the sc instruction to make hypervisor calls when the SLES system is running in a logical partition. The processor distinguishes between hypervisor calls and system calls by examining the general purpose register 0 (`GPR0`).

If the `GPR0` contains -1, and the processor is in privileged state, then the `sc` instruction is treated as a hypervisor call. Otherwise, it is treated as a system call request from user space. The `sc` instruction without -1 in GPR0 generates an exception. The exception handler in the kernel redirects the call to the `system_call()` system call handler. `system_call()` saves the contents of registers in the kernel mode stack, handles the call by invoking a corresponding C function in the kernel, and exits the handler by means of the `ret_from_sys_call()` function.

5.16.3.3 System z

On System z, z/Architecture provides the SuperVisor Call assembly instruction `SVC()` to make a system call. The SVC instruction generates an exception. The exception handler in the kernel redirects the call to the system call handler, `system_call()`. `system_call()` saves the contents of registers in the kernel mode stack, handles the call by invoking a corresponding C function in the kernel, and exits the handler by means of the `ret_from_sys_call()` function.

5.16.3.4 eServer 326

The AMD Opteron processors differ significantly from x86 architecture with respect to the entry point into the kernel. The Opteron processor provides the `SYSCALL` and `SYSRET` special instructions instead of using the interrupt 0x80. The `SYSCALL` assembly instruction performs a call to the `system_call()` system call handler running at CPL level 0.

The address of the target procedure is specified implicitly through Model Specific Registers (MSRs). `system_call()` saves the contents of registers in the kernel mode stack, handles the call by invoking a corresponding C function in the kernel, and executes the `SYSRET` privileged instruction to return control back to the user space.

5.16.4 System call argument verification

The process of transferring control from user mode to kernel mode does not generate any user-accessible objects; thus, there are no object reuse issues to handle. However, because system calls often require input parameters, which may consist of addresses in the address space of the user-mode process, an illegal access violation can occur as a result of a bad parameter. For example, a user-mode process might pass an address belonging to the kernel-address space as a parameter, and because the kernel routines are able to address all pages present in memory, the address is able to read or write any page present in the memory without causing a page fault exception. The SLES kernel prevents these kinds of access violations by validating addresses passed as system-call parameters. For the sake of efficiency, and satisfying the access control requirement, the SLES kernel performs validation in a two-step process, as follows:

1. Verifies that the linear address (virtual address for System p and System z) passed as a parameter does not fall within the range of interval addresses reserved for the kernel. That is, that the linear address is lower than `PAGE_OFFSET`.

2. Because bad addresses lower than `PAGE_OFFSET` cause a page fault, the kernel consults the exception table and verifies that the address of the instruction that triggered the exception is NOT included in the table. Exception tables are automatically generated by the C compiler when building the kernel image. They contain addresses of instructions that access the process address space.

6 Mapping the TOE summary specification to the High-Level Design

This chapter provides a mapping of the security functions of the TOE summary specification to the functions described in this High-Level Design document.

6.1 Identification and authentication

Section 5.11 provides details of the SLES Identification and Authentication subsystem.

6.1.1 User identification and authentication data management (IA.1)

Section 5.11.2 provides details of the configuration files for user and authentication management.

Section 5.11.3.6 explains how a password can be changed.

6.1.2 Common authentication mechanism (IA.2)

Section 5.11.1 provides a description of PAM, which is used to implement the common authentication mechanism for all the activities that create a user session.

6.1.3 Interactive login and related mechanisms (IA.3)

Section 5.11.3.3 provides a description of the interactive login process. Section 5.12.2 describes the process of obtaining a shell from the remote system.

6.1.4 User identity changing (IA.4)

Section 5.11.3.7 provides a description of changing identity on the local system using the `su` command.

6.1.5 Login processing (IA.5)

Section 5.11.3.3 provides details of the `login` process and also a description of changing identity on the local system.

6.2 Audit

Section 5.6 provides details of the Linux audit subsystem.

6.2.1 Audit configuration (AU.1)

Section 5.6.2 provides details of configuration of the audit subsystem to select events to be audited based on rules defined in `/etc/audit.rules` audit configuration file. Section 5.15.3 describes how configuration parameters are loaded into the SLES kernel.

6.2.2 Audit processing (AU.2)

Sections 5.6.1 and 5.6.1.2 provide details of how processes attach and detach themselves from the audit subsystem. Section 5.15.1 describes the audit daemon and how it reads audit data from the kernel buffer and writes audit records to a disk file.

6.2.3 Audit record format (AU.3)

Section 5.6.3.2 describes information stored in each audit record.

6.2.4 Audit post-processing (AU.4)

Section 5.15.2 describes audit subsystem utilities provided for post-processing of audit data.

6.3 Discretionary Access Control

Sections 5.1 and 5.2 provide details on Discretionary Access Control (DAC) on the SLES system.

6.3.1 General DAC policy (DA.1)

Sections 5.1 and 5.2.2 provides details on the functions that implement general Discretionary Access policy.

6.3.2 Permission bits (DA.2)

Sections 4.1.2.1.2, 4.1.2.1.3, 5.1.2.1, 5.1.5.1, and 5.11.2.1 provide details on calls that perform DAC based on permission bits.

6.3.3 ACLs (DA.3)

Sections 5.1.2.1, 5.1.5.2, and 5.1.5.2.1 provide details on DAC based on ACLs on file system objects.

6.3.4 DAC: IPC objects (DA.4)

Section 5.3 provides details on DAC for IPC objects.

6.4 Object reuse

Sections 5.1, 5.2, 5.3, and 5.5 provide details on object reuse handling by the SLES kernel.

6.4.1 Object reuse: file system objects (OR.1)

Section 5.1.2.1 provides details on object reuse handling for data blocks for file system objects.

6.4.2 Object reuse: IPC objects (OR.2)

Sections 5.3.3.2, 5.3.3.3, 5.3.3.4, and 5.3.3.5 provide details on object reuse handling for message queues, semaphores, and shared-memory segments.

6.4.3 Object reuse: memory objects (OR.3)

Sections 5.5.2.1, 5.5.2.2, and 5.5.2.4 provide details on object reuse handling for memory objects.

6.5 Security management

Section 5.13 provides details about various commands used to perform security management.

6.5.1 Roles (SM.1)

Section 5.13 provides details on various commands that support the notion of an administrator and a normal user.

6.5.2 Access control configuration and management (SM.2)

Sections 5.1.1 and 5.1.2.1 provide details on the system calls of the file system that are used to set attributes on objects to configure access control.

6.5.3 Management of user, group and authentication data (SM.3)

Sections 5.11.2 and 5.13 provide details on various commands used to manage authentication databases.

6.5.4 Management of audit configuration (SM.4)

Sections 5.15.1 and 5.15.2 describe utilities used to upload audit configuration parameters to the SLES kernel and utilities used by trusted processes to attach and detach from the audit subsystem.

6.5.5 Reliable time stamps (SM.5)

Sections 3.1.1, 3.2.1, 3.3.1, and 3.4.1 describe the use of hardware clocks, by eServer hardware, to maintain reliable time stamps.

6.6 Secure communications

Sections 5.12.1 and 5.12.2 describe secure communications protocols supported by SLES.

6.6.1 Secure protocols (SC.1)

Section 5.12.2 describes the Secure Shell (SSH) protocol. Section 5.12.1 describes the Secure Socket Layer (SSL) protocol. Section 5.12.1.3 describes cipher suites and cryptographic algorithms supported by SLES.

6.7 TSF protection

Chapter 4 provides details on TSF protection.

6.7.1 TSF invocation guarantees (TP.1)

Section 4.2 provides details of the TSF structure. Section 4.2 also provides a mechanism to separate TSF software from non-TSF software.

6.7.2 Kernel (TP.2)

Section 4.2.1 provides details on the SLES kernel.

6.7.3 Kernel modules (TP.3)

Section 4.2.1.2 provides details on kernel modules on the SLES system.

6.7.4 Trusted processes (TP.4)

Section 4.2.2 provides details on the non-kernel trusted process on the SLES system.

6.7.5 TSF Databases (TP.5)

Section 4.3 provides details on the TSF databases on the SUSE Linux Enterprise Server system.

6.7.6 Internal TOE protection mechanisms (TP.6)

Section 4.1.1 describes hardware privilege implementation for the System x, System p, System z and Opteron eServer 326. Section 5.5 describes memory management and protection. Section 5.2 describes process control and management.

6.7.7 Testing the TOE protection mechanisms (TP.7)

Section 5.13 describes the AMTU tool available to administrative user to test the protection features of the underlying abstract machine.

6.8 Security enforcing interfaces between subsystems

This section identifies the security enforcing interfaces between subsystems in the high level design of SLES. The individual functions exported by each subsystem are described with the subsystem itself. This section therefore only discusses in general how the individual subsystems work together to provide the security functions of the TOE. This section is mainly used to identify those internal interfaces between subsystems that are security enforcing in the sense that the subsystems work together to provide a defined security function. Interfaces that are not security enforcing are interfaces between subsystems where the interface is not used to implement a security function.

There is also the situation where a kernel subsystem A invokes functions from another kernel subsystem B using the external interface of the kernel subsystem. This, for example, is the case when a kernel subsystem needs to open and read or write files, using the File and I/O kernel subsystem, or when a kernel subsystem sets the user ID or group ID of a process, using the Process Control subsystem.

In those cases, all the security checks performed by those interface functions apply. Note that a system call function in the kernel operates with the real and effective user ID and group ID of the caller unless the kernel function that implements the system call changes this.

This section describes the interfaces between subsystems, but it only discusses interfaces between kernel components that directly implement security functions. Note that kernel subsystems can use the kernel internal interfaces described in the individual subsystems as well as the externally visible interfaces (system calls).

The subsystems are:

- Kernel subsystems:
  - File and I/O
  - Process Control
  - Inter-Process Communication
  - Networking
  - Memory Management
  - Audit

- Kernel Modules
- Device Drivers

- Trusted process subsystems:
  - System Initialization
  - Identification and Authentication
  - Network Applications
  - System Management
  - Batch Processing
  - User-level audit subsystem

6.8.1 Summary of kernel subsystem interfaces

This section identifies the kernel subsystem interfaces and structures them per kernel subsystem into:

External Interfaces: System calls associated with the various kernel subsystems form the external interfaces. They are structured into TSFI System Calls and Non-TSFI System Calls.

Internal Interfaces: These are the interfaces that cannot be exported as system calls that are intended to be used by other kernel subsystem. Note that other kernel subsystems may of course also use the system calls by calling the kernel internal entry point of the system call. This entry point can be the name of the system call prefixed with sys_, or the name of the system call, prefixed with ppcX_ (X can be 32 or 64) or ppc_ (PowerPC kernel). For example, for a system call abc, the kernel internal entry point is either sys_abc or ppcX_abc or ppc_abc.

Data Structures: The kernel subsystem maintains data structures that can be read directly by other kernel subsystems to obtain specific information. They are considered to be data interfaces. Data structures are defined in header files.

The system calls are not described any further in this chapter. For more information about the purpose of the system call, its parameter, return code, restrictions, and effects, see the man page for that particular system call. The spreadsheet delivered as part of the functional specification also shows on which platform the different system calls are available.

This chapter contains a reference to the internal interfaces, describing where to find the description of the function implementing this internal interface.

Concerning the data structures, this chapter contains the name of the header file within the TOE source tree that defines the data structure. This document, as well as the other documents provided as references within this chapter, provides details of the purpose of those data structures.

6.8.1.1 Kernel subsystem file and I/O

This section lists external interfaces, internal interfaces, and data structures of the file and I/O subsystem.

6.8.1.1.1 External Interfaces

TSFI system calls

Non-TSFI system calls

System calls are listed in the Functional Specification mapping table.

6.8.1.1.2 Internal Interfaces

6.8.1.1.3

Internal function	**Interfaces defined in**
`permission`	This document, Section 5.1.1.1
`vfs_permission`	This document, Sections 5.1.1.1 and 5.1.5.1
`get_empty_filp`	This document, Section 5.1.1.1
`fget`	This document, Section 5.1.1.1
`do_mount`	This document, Section 5.1.2.1

Specific ext3 methods	**Interfaces defined in**
`ext3_create`	This document, Section 5.1.2.1
`ext3_lookup`	This document, Section 5.1.2.1
`ext3_get_block`	This document, Section 5.1.2.1
`ext3_permission`	This document, Sections 5.1.2.1, 5.1.5.1, and 5.1.5.2
`ext3_truncate`	This document, Section 5.1.2.1

Specific isofs methods	**Interfaces defined in**
`isofs_lookup`	This document, Section 5.1.2.2

Basic inode operations. The create through the revalidate operations are described in Section 5.1.1, attribute and extended attribute functions are described in this document in Section 5.1.2.1 in the context of the ext3 file system.

`create`	`followlink`
`lookup`	`truncate`
`link`	`permission`
`unlink`	`revalidate`
`symlink`	`setattr`
`mkdir`	`getattr`
`rmdir`	`setxattr`
`mknod`	`getxattr`
`rename`	`listxattr`
`readlink`	`removexattr`

Inode: note that these are super-user operations, which be used by other subsystems, so there is no subsystem interface:

`read_inode`	`write_super`
`read_inode2`	`write_super_lockfs`
`dirty_inode`	`unlockfs`
`write_inode`	`statfs`
`put_inode`	`remount_fs`
`delete_inode`	`clear_inode`

Dentry operations: Note that they are not used by other subsystems, so there is no subsystem interface:

- d_revalidate
- d_hash
- d_compare
- d_delete
- d_release
- d_iput

6.8.1.1.4 Data Structures

`super_block`	`include/linux/fs.h`
`ext3_super_block`	`include/linux/ext3_fs.h`
`isofs_sb_info`	`include/linux/iso_fs_sb.h`
`inode`	`include/linux/fs.h`
`ext3_inode`	`include/linux/ext3_fs.h`
`iso_inode_info`	`include/linux/iso_fs_i.h`
`ext3_xattr_entry`	`include/linux/ext3_xattr.h`
`file`	`include/linux/fs.h`
`dentry`	`include/linux/dcache.h`
`vfsmount`	`include/linux/mount.h`

6.8.1.2 Kernel subsystem process control and management

This section lists external interfaces, internal interfaces, and data structures of the process control and management subsystem.

6.8.1.2.1 External interfaces (system calls)

TSFI system calls

Non-TSFI system calls

System calls are listed in the Functional Specification mapping table.

6.8.1.2.2 Internal Interfaces

Internal function	**Interfaces defined in**
`current`	*Understanding the LINUX KERNEL*, Chapter 3, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0
`request_irq`	*Linux Device Drivers*, O'Reilly, Chapter 9, 2nd Edition June 2001, Alessandro Rubini
`free_irq`	*Linux Device Drivers*, O'Reilly, Chapter 9, 2nd Edition June 2001, Alessandro Rubini
`send_sig_info && check_kill_permission`	*Understanding the LINUX KERNEL*, Chapter 10, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0/ and this document, chapter 5.3.4.2

6.8.1.2.3 Data Structures

`task_struct` and `include/linux/sched.h`

6.8.1.3 Kernel subsystem inter-process communication

This section lists external interfaces, internal interfaces, and data structures of the inter-process communication subsystem.

6.8.1.3.1 External interfaces (system calls)

- TSFI system calls
- Non-TSFI system calls

System calls are listed in the Functional Specification mapping table.

6.8.1.3.2 Internal Interfaces

Internal function	Interfaces defined in
`do_pipe`	*Understanding the LINUX KERNEL*, Chapter 19, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0/ and this document, Section 5.3.1.1
`pipe_read`	*Understanding the LINUX KERNEL*, Chapter 19, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0/ and this document, Section 5.3.1.1
`pipe_write`	*Understanding the LINUX KERNEL*, Chapter 19, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0/ and this document, Section 5.3.1.1
`init_special_inode`	This document, Section 5.3.2.1
`fifo_open`	This document, Section 5.3.2.2
`ipc_alloc`	This document, Section 5.3.2.2
`ipcperms`	This document, Section 5.3.2.2
`send_sig_info`	This document, Section 5.3.2.2
`unix_create`	This document, Section 5.3.5
`inet_create`	This document, Section 5.3.5
`sk_alloc`	This document, Section 5.3.5

6.8.1.3.3 Data Structures

`ipc_ids`	`ipc/util.h`
`ipc_id`	`ipc/util.h`
`kern_ipc_perm`	`include/linux/ipc.h`
`msg_queue`	`ipc/msg.c`
`msg_msg`	`ipc/msg.c`
`sem_array`	`include/linux/sem.h`
`shmid_kernel`	`ipc/shm.c`
`sock`	`include/net/sock.h`

6.8.1.4 Kernel subsystem networking

This section lists external interfaces, internal interfaces and data structures of the networking subsystem.

6.8.1.4.1 External interfaces (system calls)

- TSFI system calls
- Non-TSFI system calls

System calls are listed in the Functional Specification mapping table.

6.8.1.4.2 Internal interfaces

Sockets are implemented within the inode structure as specific types of inodes. `inode.u`, in the case of an inode for a socket, points to a structure of type socket. This structure contains a pointer to the `proto_ops struct`, in which the following methods of the socket are included:

`release`	`listen`
`bind`	`shutdown`
`connect`	`setsockopt`
`socketpair`	`getsockopt`
`accept`	`sendmsg`
`getname`	`recvmsg`
`poll`	`mmap`
`ioctl`	`sendpage`

The socket system call creates the inode. The system calls, such as `bind`, `connect`, `poll`, `listen`, `setsockopt`, `getsockopt`, `ioctl`, and `accept`, are directly implemented by the methods registered for the socket.

`read` and write, as well as `send`, `sendmsg`, `sendto` and `recv`, `recvfrom`, and `recvmsg` are implemented by the methods registered for `sendmsg` and `recvmsg`. Close is implemented by the methods registered for shutdown and release, and `getsockname` is implemented by the method registered for `getname`. Please note that send is an alias for `sendmsg`, and `recv` is an alias for `recvfrom`.

6.8.1.4.3 Data Structures

`socket`	`include/linux/net.h`
`proto_ops`	`include/linux/net.h`

6.8.1.5 Kernel subsystem memory management

This section lists the external interfaces, internal interfaces and data structures of the memory management subsystem.

6.8.1.5.1 External interfaces (system calls)

- TSFI system calls
- Non-TSFI system calls

System calls are listed in the Functional Specification mapping table

6.8.1.5.2 Internal interfaces

Internal interfaces	Interfaces defined in
`get_zeroed_page`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini /this document, chapter 5.5.2.1
`__vmalloc`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini
`vfree`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini
`kmalloc`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini
`kfree`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini
`__ get_free_pages`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini
`free_pages`	*Linux Device Drivers*, O'Reilly, Chapter 7, 2nd Edition June 2001, Alessandro Rubini

6.8.1.5.3 Data Structures

`mm_struct` and `include/linux/sched.h`

6.8.1.6 Kernel subsystem audit

This section lists external interfaces, internal interfaces, and data structures of the audit subsystem.

6.8.1.6.1 External interfaces

There are two external interfaces to the audit subsystem.

- Netlink socket calls, by which all kernel user-space communication takes place.
- Communication through `/proc` to associate the login uid with the login session and therefore with all tasks forked or exec'ed under the session.

6.8.1.6.2 Internal interfaces

The audit kernel provides a set of interfaces to other kernel subsystems to:

- Format and send audit records to user space, audit_log_* functions.
- Allocate per task audit context, `audit_alloc` which is called by `copy_proc`.
- Audit syscalls on entry and exit, `audit_syscall_exit` and `audit_syscall_entry`.
- Audit file system watched objects, `audit_notify_watch` and `audit_notify_update`.
- Add additional audit information for specific audit events:
    - audit_socketcall

- audit_sockaddr
- audit_ipc_perms

6.8.1.6.3 Data structures

- `audit_sock`: The netlink socket through which all user space communication is done.
- `audit_buffer`: The audit buffer is used when formatting an audit record to send to user space. The audit subsystem pre-allocates audit buffers to enhance performance.
- `audit_context`: The audit subsystem extends the task structure to potentially include an `audit_context`. On task creation, by default the audit context is built, unless specifically denied by the per-task filter rules. The audit subsystem further extends the `audit_context` to allow for more auxiliary audit information that may be needed for specific audit events. While the auxiliary information is collected during the execution of the system call, all other `audit_context` information is filled on entry and exit of the system call.
- `audit_filter_list`: An array of filter lists to store various filter rules for various type of filters. Current filters supported are task creation time filter, syscall entry filter, syscall exit filter, file system watch filter, and user message filter.
- `audit_watch`: A structure that holds audit information associated with an inode to watch.
- `watchlist`: Per-directory list associated with directories that have watched children.
- `master_watchlist`: A linked list that holds all the watches in the system.
- `auditfs_hash_table`: A hash table of hashed inode addresses to store and retrieve inode audit data.

6.8.1.7 Kernel subsystem device drivers

6.8.1.7.1 External interfaces (system calls)

No direct interface.

- Device driver-specific commands can be passed from a user-space program to the device driver using the ioctl system call, which is a system call of the File and I/O subsystem.
- File and I/O first checks for some generic ioctl commands it can handle itself, and calls the device driver ioctl method if the request by the user program is not one of those generic requests. To issue an ioctl system call, the calling process must first have opened the device, which requires full access rights to the device itself. Those access checks are performed by the open system call within the File and I/O subsystem.

6.8.1.7.2 Internal interfaces

Device drivers implement a set of methods that other kernel subsystems can directly use. In most cases, the File and I/O subsystem will use those methods after the processing of a user's request, including checking the user's right to perform the requested action. The internal interfaces are therefore the methods implemented by the various device drivers.

All checks, according to the security policy, have to be performed by the kernel subsystem, invoking a method of a specific device driver before it calls the function. For a description of the purpose of the device driver methods for character device drivers and block device drivers, see [RUBN]. Chapter 3 describes the methods for character devices and chapter 6 describes the methods for block devices.

6.8.1.7.2.1 Character Devices

Possible Character Device methods are:

`llseek`	`flush`
`read`	`release`
`write`	`fsync`
`readdir`	`fasync`
`poll`	`lock`
`ioctl`	`readv`
`mmap`	`writev`
`open`	`owner`

Other functions:

`register_chrdev`	*Linux Device Drivers, O'Reilly*, Chapter 3, 2nd Edition June 2001, Alessandro Rubini
`unregister_chrdev`	*Linux Device Drivers, O'Reilly*, Chapter 3, 2nd Edition June 2001, Alessandro Rubini

6.8.1.7.2.2 Block Devices

Possible Block Device Methods are:

- open
- release
- ioctl
- check_media_change
- revalidate

In addition to this, the programmer needs to define the device-specific function request().

Other functions include the following:

`register_blkdev`	*Linux Device Drivers*, O'Reilly, Chapter 12, 2nd Edition June 2001, Alessandro Rubini
`unregister_blkdev`	*Linux Device Drivers*, O'Reilly, Chapter 12, 2nd Edition June 2001, Alessandro Rubini

6.8.1.7.3 Data structures

device_struct	fs/devices.c
file_operations	include/linux/fs.h
block_device_operati ons	include/linux/fs.h

6.8.1.8 Kernel subsystems kernel modules

This section lists external interfaces, internal interfaces, and data structures of the kernel modules subsystem.

6.8.1.8.1 External interfaces (system calls)

- TSFI system calls
- Non_TSFI system calls

System calls are listed in the Functional Specification mapping table.

6.8.1.8.2 Internal interfaces

Module dependent.

6.8.1.8.3 Data structures

Module dependent.

6.8.2 Summary of trusted processes interfaces

Trusted processes need to use system calls when they need the functions of a kernel subsystem. The interfaces to the kernel subsystems, therefore, are only the system calls. Trusted processes can communicate with each other using the named objects provided by the kernel, which are files and IPC objects. There is no way for trusted processes to communicate with other without using those primitives provided by the kernel.

As described in the functional specification, trusted processes use configuration files as an external interface used to define their behavior. Those configuration files are described as man pages in the functional specification, and their use by the trusted processes is described in this document in the sections about the individual trusted processes.

7 References

[CC] *Common Criteria for Information Technology Security Evaluation, CCIMB-99-031, Version 2.1, August 1999*

[CEM] *Common Methodology for Information Technology Security Evaluation, CEM-99/045, Part 2 – Evaluation Methodology, Version 1.0, 1999*

[BOVT] *Understanding the LINUX KERNEL*, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0

[ROBT] *Linux Kernel Development - A Practical guide to the design and implementation of the Linux Kernel* by Robert Love

[MANN] *Linux System Security*, 2nd Edition, Scott Mann, Ellen Mitchell, Mitchell Krell, ISBN# 0-13-047011-2

[OF94] *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements*

[STALLS] *Cryptography and Network Security*, 2nd Edition, William Stallings, ISBN# 0-13- 869017-0

[LH] *Linux Handbook, A Guide to IBM Linux Solutions and Resources*, Nick Harris et al.

[PSER] *IBM eServer pSeries and IBM RS6000 Linux Facts and Features*

[ZPOP] *z/Architecture Principles of Operation*

[COMR] *Internetworking with TCP/IP*, Douglas E. Comer & David L. Stevens, ISBN# 0-13-474222-2

[RODR] *TCP/IP Tutorial and Technical Overview*, Adolfo Rodriguez, et al.

[EYNG] *Internet Security Protocols: SSLeay & TLS*, Eric Young

[DRKS] *The TLS Protocol version 1*, Tim Dierks, Eric Rescorla

[ENGD] *PowerPC 64-bit Kernel Internals*, Engebretsen David

[DBOU] *The Linux Kernel on iSeries*, David Boutcher

[TBEI] *Linux Audit Subsystem Design Documentation*, Thomas Biege, Emily Ratliff and Daniel Jones

[ALM] *Booting Linux: History and the Future*, 2000 Ottawa Linux Symposium, Almesberger, Werner.

[KFEN] Linux Security HOWTO, Kevin Fenzi, Dave Wreski

[BURG] Security-QuickStart HOWTO, Hal Burgiss

[RATL] Abstract Machine Testing: Requirements and Design, Emily Ratliff

[INTL] *Intel Architecture Software Developer's Manual Volume 3: System Programming*

[AMD64] *AMD64 Architecture, Programmer's Manual Volume 2: System Programming*

[AKLN] *Porting Linux to x86-64*, Andi Kleen

[ALTM] *The value of z/VM: Security and Integrity*, Alan Altmark and Cliff Laking

[ZGEN] z/VM general Information

[IINIT] *LPAR Configuration and Management – Working with IBM eServer iSeries Logical Partitions*

[ZINIT] *Linux on IBM eServer zSeries and S/390: Building SuSE SLES8 Systems under z/VM*

[RUBN] *Linux Device Drivers*, O'Reilly, 2nd Edition June 2001, Alessandro Rubini

[RSA] "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, Feb 1978, pp. 120-126, R. Rivest, A. Shamir, and L. M. Adleman,

[DH1] "New Directions in Cryptography," *IEEE Transactions on Information Theory*, V.IT-22, n. 6, Jun 1977, pp. 74-84, W. Diffie and M. E. Hellman.

[DSS] NIST FIPS PUB 186, "Digital Signature Standard," National Institute of Standards and Technology, U.S.Department of Commerce, 18 May 1994.

[SCHNEIR] "Applied Cryptography Second Edition: protocols algorithms and source in code in C",1996, Schneier, B.

[FIPS-186] Federal Information Processing Standards Publication, "FIPS PUB 186, Digital Signature Standard", May 1994.

[CRISP] *SubDomain: Parsimonious Server Security* by Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor at https://forgesvn1.novell.com/viewsvn/apparmor/trunk/docs/papers/subdomain-lisa00.pdf?revision=3.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. Support regarding capabilities of non-IBM products should be addressed to the suppliers of those products.

Any statements about support or other commitments may be changed or canceled at any time without notice. All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. This Information is provided "AS IS" without warranty of any kind.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

The information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.