SLES High Level Design

Version 3.16

Change Log

Version	Date	Authors	Reviewers	Changes, Problems, Notes
3.0	5/1/04	JRD		Original version based on EAL3 HLD.
3.1	5/24/04	JRD		Added squadron info
3.2	5/25/04	JRD		Rtas, sysfs and cleanup
3.3	6/01/04	JRD		Removed UnitedLinux references
3.4	6/27/04	JRD		Updated to sync up with the latest functional specification.
3.5	7/8/04	JRD	HK	Updated to incorporate Helmut's feedback
3.6	7/13/04	JRD	HK	Updated to incorporate Helmut's feedback
3.7	7/19/04	JRD	HK	Updated to incorporate Helmut's feedback regarding reference information for V5 R3
3.8	7/25/04	JRD	HK	Updated system call handler name
3.9	7/26/04	JRD	HK	Updates to incorporate Helmut's feedback
3.10	8/5/04	JRD	SM	Added atrm, /var/log/faillog as per Stephan's feedback
3.11	9/29/04	JRD	SM	Updated as per Stephan's feedback – corrected section names in 6.5.4
3.12	11/15/04	JRD	SM	Added reference to POWER5 in section 5.5.1.2
3.13	11/16/04	JRD	SM	Added missing audit interfaces in section 6.8.1.6.2
3.14	11/18/04	JRD	HK	Updated package list. Clarified that POWER5 pSeries is used only in LPAR mode.
3.15	11/20/04	JRD	SM	Added description of tmpfs
3.16	11/29/04	JRD	HK	Some minor updates: added atrm description in chapters 4 and 5, special handling of device special files in read only file systems, added the additional cipher suites supported by the SSL implementation of the TOE and explained that the boot process for pSeries also applies for booting the TOE in a.logical partition.

1 Introduction

This document, the SLES High Level Design (HLD), summarizes the design and Target of Evaluation Security Functions of the SUSE® Linux® Enterprise Server (SLES) Operating System version 9. This document is used within the Common Criteria evaluation of SLES at Evaluation Assurance Level (EAL) 4 and describes the security functions defined in the Common Criteria Security Target document. Other security-related functions of SLES that are not available or used in the evaluated configuration are not described in this document.

1.1 Purpose of this document

The SLES HLD is a high-level design document that summarizes the design of the product and provides references to other, more detailed, design documentation. The SLES HLD is consistent with additional high-level design documents (pointers to those documents are included), as well as with the supporting detailed design documents for the system.

The SLES HLD is intended for the evaluation team as a source of information about the architecture of the system. The document provides pointers to detailed design documentation regarding the structure of and functions performed by the system.

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 Target of Evaluation Security Functions (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 TOE summary specification from the SLES Security Target to specific sections in this document.

1.3 Conventions used in this document

The following notational conventions are used in this document:

`Constant Width`
: Shows the contents of code files or output from commands, and indicates source-code keywords that appear in code.

Italic
: Used for file and directory names, program and command names, command-line options, URLs, and for emphasizing new terms.

1.4 Terminology

This section contains definitions of technical terms that have specific definitions for the purposes of this document. Terms defined in the [CC] are not reiterated here unless stated otherwise.

SLES: An abbreviation of "SUSE Linux Enterprise Server 9."

Administrative user: An administrator of a SUSE Linux Enterprise Server system. Some administrative tasks require the use of the *root* username and password so that they can become the superuser (with a user ID of 0).

Authentication data: Includes a user identifier, password, and authorizations for each user of the product.

Object: In SLES, objects belong to one of three categories: file system objects, IPC objects, and memory objects.

Product: Defines software components that comprise the distributed SLES system.

Public object: A type of object for which all subjects have read access, but only the TSF or the system administrator have write access.

Role: Represents a set of actions that an authorized user, upon assuming the role, can perform.

Security attributes: As defined by functional requirement FIA_ATD.1, includes the following as a minimum: user identifier, group memberships, and user authentication data.

Subject: There are two classes of subjects in SLES:

> *untrusted internal subject*, which is a SLES process running on behalf of some user outside of the TSF (for example, with no privileges).
>
> *trusted internal subject*, which is a SLES process running as part of the TSF (for example, service daemons and the process implementing the identification and authentication of users).

System: Includes the hardware, software, and firmware components of the SLES product that are connected or networked together and configured to form a usable system.

Target of Evaluation (TOE): The SUSE Linux Enterprise Server version 9 operating system, running and tested on the hardware specified in this High Level Design document. The BootPROM firmware and the hardware form part of the TOE Environment.

User: Any individual who has a unique user identifier and who interacts with the SLES product.

2 System overview

The Target of Evaluation (TOE) is SLES version 9 running on an IBM eServer host computer. Multiple TOE systems can be connected via a physically protected Local Area Network (LAN). The IBM eServer line consists of Intel® processor-based xSeries® systems, POWER4+™ and POWER5™ processor-based pSeries® and iSeries™ systems, IBM mainframe zSeries® systems, and AMD Opteron processor-based systems that are intended for use as networked workstations and servers.

The following figure shows a series of interconnected TOE systems. Each TOE system is running the SLES version 9 operating system on an eServer computer. SLES version 9 is based on the globally available Linux operating system for enterprise customers.

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 potentially from other computers within the LAN.

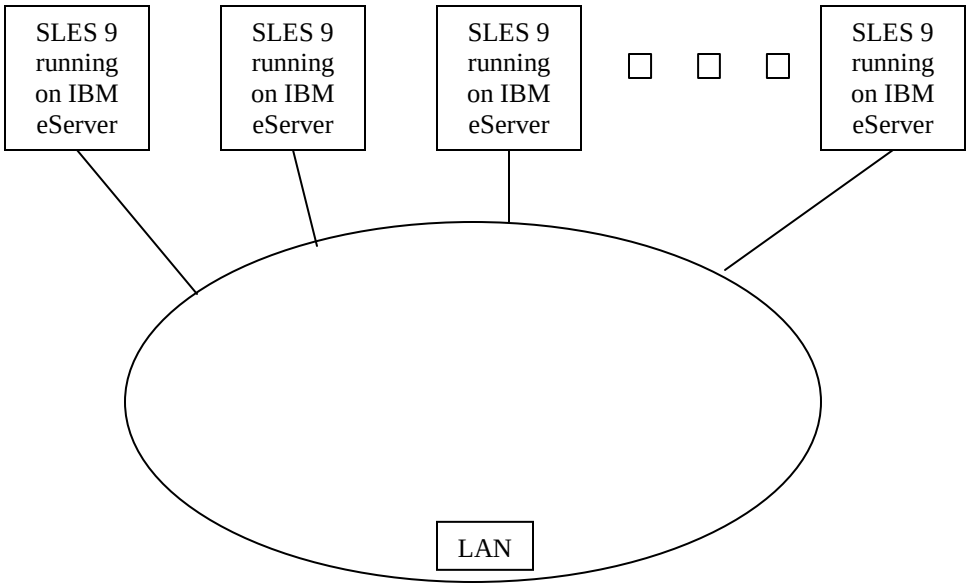

Figure 2-1. Series of TOE systems connected by a physically protected LAN

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 *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) by requiring each user to log in with the 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 SLES and 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 SUSE Linux Enterprise Server and the IBM eServer series systems.

2.1.1 SUSE Linux Enterprise Server (SLES)

SUSE Linux Enterprise Server is based on version 2.6.5 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 IBM eServer systems

In 2000, IBM introduced the IBM eServer, a new generation of servers featuring mainframe-class reliability and scalability, broad support of open standards for the development of new applications, and capacity on demand for managing the unprecedented needs of e-business. The new servers feature technology from IBM's high-end servers applied across the entire product line and include:

- eServer zSeries – the reliable, mission-critical data and transaction server
- eServer pSeries -- the powerful, technologically advanced POWER4+ and POWER5 processor-based server
- eServer iSeries -- the high performance, POWER4+ and POWER5 integrated business server for mid-market companies
- eServer xSeries -- the affordable Intel-based server with mainframe-inspired reliability technologies
- eServer 325 – the AMD Opteron-based server with outstanding value in high performance computing in both 32-bit and 64-bit environments.

Since introducing eServers in 2000, new models with more powerful processors were added to the xSeries, pSeries, iSeries, and zSeries lines of systems. The AMD Opteron processor-based eServer 325 was added to the eServer series in 2003. The AMD Opteron eServer 325 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 Target of Evaluation consists of SLES running on an eServer computer. The TOE system can be connected to other systems by a protected Local Area Network.

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 developing their own software to run in their own protected environment.

The Common Criteria for Information Technology Security Evaluation (CC) and the Common Methodology for Information Technology Security Evaluation (CEM) call for 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 up 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 (for example, 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 the hardware itself is part of the TOE environment, but 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 the following figure, the system consists of eServer hardware, the SLES kernel, trusted non-kernel processes, TSF databases, and untrusted processes. The TOE itself consists of the Kernel Mode Software and User Mode Software. 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 video frame buffer are omitted.)

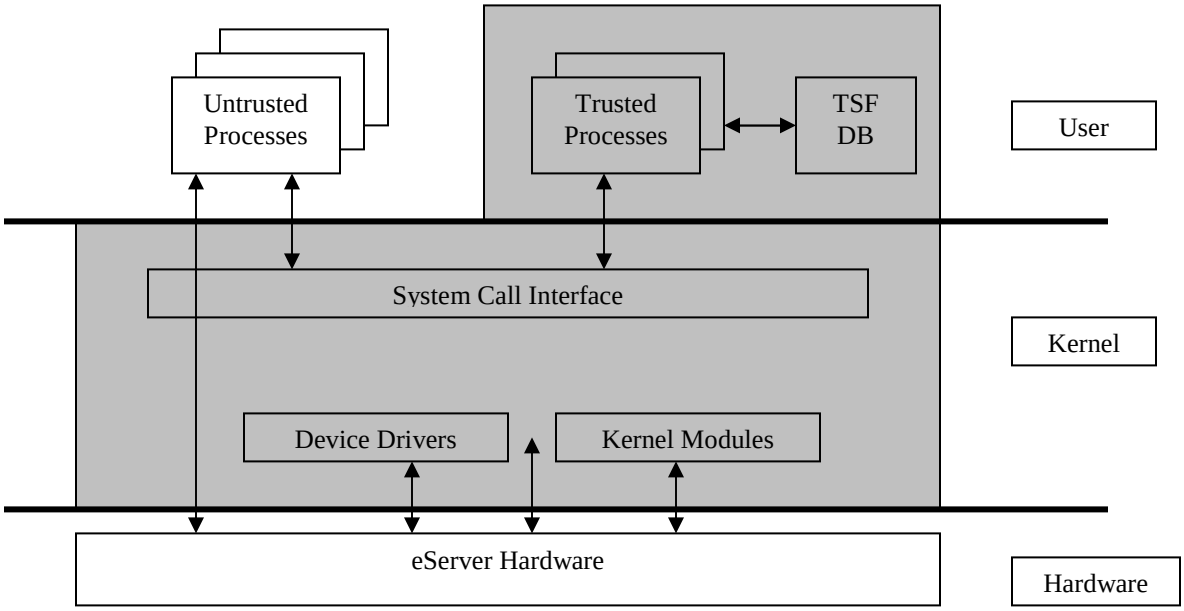

Figure 2-2. Overall structure of the TOE

The hardware includes the planar (CPUs, memory, buses, onboard adapters, and support circuitry), additional adapters (for example, LAN and video), and other peripherals (storage devices, monitors, keyboards, and front-panel hardware).

The SLES kernel includes separately loadable kernel modules and device drivers (a device driver can also be a kernel module) in addition to the base SLES kernel. The kernel consists of the bootable kernel image and its loadable modules. The kernel implements the SLES 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. The structure of the SLES kernel is described further in the Software Architecture chapter.

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

The SLES system (hardware, kernel mode software, non-kernel programs, and databases) provides an environment in which users and administrators run programs (sequences of CPU instructions) within a protected environment. Programs execute as processes with the identity of the user that started them (except for some exceptions defined in this HLD) and are subject to the system's access control and accountability policies.

2.2.2 eServer system structure

The system consists of an eServer computer described previously, which permits one user at a time to log in to a computer's console. Several virtual consoles can be mapped to a physical console. Different users can login through different virtual consoles simultaneously. The system can be connected to other computers via physically and logically protected Local Area Networks (LANs). The eServer hardware and the physical LAN interconnecting the different systems running SLES are not included within the evaluation boundary. External routers, bridges, and repeaters are also not included in the evaluation boundary.

A standalone host configuration operates as a CC-evaluated system, used by multiple users at a time. Users can operate by logging in at the system's virtual consoles or serial terminals, 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. The networking software is loaded, even though there may be no other hosts on the network. When the networking software is loaded, users can request network services (for example, FTP) from server processes on the same host.

Another configuration provides a useful network configuration on 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 transfer files from one host to another. The configuration extends the single LAN architecture to show that SLES provides 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 one LAN segment, and they are routed by a host in this segment to a host on a second LAN segment. The packets are eventually routed by a host in the second LAN segment to a host on a third LAN segment, where they 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 transparent to the user (except for performance). 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.

Only the computers and software described in this document can be included in the TOE. This means that none of the TOE computers in the LAN can be permitted to provide dialup connections, or can be connected to the external Internet.

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 local computer console, virtual consoles, or terminal devices connected through physically protected serial lines.
- Local services to previous users via deferred jobs (for example, *at* and *cron*).
- Local services to users who have accessed the local host via the network using a protocol such as *ssh* that starts a user shell on the local host.
- Network services to potentially multiple users on either the local host or on remote hosts.

The following figure illustrates the difference between local services, which take place on each local host computer, and network services, which involve a 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 local host's kernel. All such local services take place solely on the local host computer and are mediated solely by trusted software on that host.

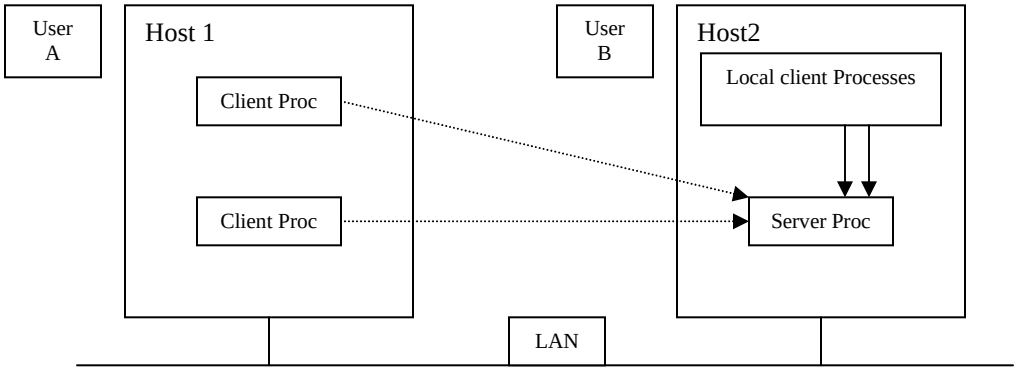

Figure 2-3. Local and network services provided by SLES systems

Network services, such as *ssh* or *ftp*, on the other hand, involve a client-server architecture and a network service-layer protocol. The client-server model splits up 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 (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.

Note also that the network client and server may 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: either by interacting directly with the system via a session at a computer console (in which case the user can use the graphical display provided as the console), by interacting directly with system via a session at a serial terminal, or through deferred execution of jobs using the *cron* and *at* utilities. A user must log in at the local system in order to access the system's protected resources. 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; that is, 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. (For 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 `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 interprocess communications 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 Discretionary Access Control (DAC) policy for all named objects under its control and an object reuse policy for all storage objects under its control. 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 multi-state processor, memory segment protection, and memory page protection. The TOE software relies on these hardware mechanisms to implement TSF isolation, noncircumventability, and process address-space separation.

A user can log in at the console, at other directly attached terminals, or via a network connection. Authentication is based on a password entered by the user and authentication data stored in a protected file. A user must log in to a host before he can access any named objects on that host. Some services, such as using *ssh* to obtain a shell prompt on another host, or using *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 password lifetime.

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

2.2.5 Operation and administration

eServer networks can be composed of one, several, or many different host computers, each of which can be in various states (shut down, initializing, 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 commands *useradd, usermod,* and *userdel* to add, modify, and delete a user account. Similarly, the commands *groupadd*, *groupmod* and *groupdel* allow an administrator to add, modify and delete a group form 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 thus provide a safer way than manual editing to update authentication databases. Please refer to the appropriate command man pages and the *SLES Security Guide* for detailed information on 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 defines 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.

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. These interfaces are not TSF interfaces because the hardware is not part of the TOE. Hardware functions prohibit user processes from directly interfering with hardware resources in a way that could bypass the security functions.
- Interfaces between non-TSF elements of the system (for example, a call from a user program to a library function) are not TSF interfaces because they do not involve the trusted parts of the TOE.
- 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 high level design.

Thus, 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.

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 that runs in a privileged hardware state (kernel, kernel module, device driver).
- It enforces the system's security policy.
- It allows setuid or setgid to a privileged user (for example, root) or group.
- It started as a privileged daemon; for example, by */etc/init.d/rc*.
- 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 system's security policies:

- 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 media.
- The install software may change the configuration (for example, mode bits) so that a program cannot violate the security policy.

The following table lists the packages and their versions that are installed on TOE:

All Platforms:	
aaa_base	9-29.13
aaa_skel	2004.6.8-0.2
acl	2.2.21-54.1
amtu	0.1-1.6

All Platforms:	
ash	0.4.18-56.1
at	3.1.8-898.1
attr	2.4.12-56.1
autoyast2-installation	2.9.45-0.2
bash	2.05b-305.6
bc	1.06-744.1
bzip2	1.0.2-346.1
certification-sles-ibm-eal4	1.0
core-release	9-6.3
coreutils	5.2.1-23.5
cpio	2.5-324.1
cracklib	2.7-1006.1
cron	3.0.1-920.1
curl	7.11.0-39.1
cyrus-sasl	2.1.18-33.1
db	4.2.52-86.3
device-mapper	1.00.09-17.5
devs	9-16.8
dialog	0.9b-188.1
diffutils	2.8.4-75.1
dosfstools	2.10-90.1
e2fsprogs	1.34-115.1
ed	0.2-864.1
evms	2.3.3-0.15
file	4.07-48.5
filesystem	9-29.4
fillup	1.42-98.1
findutils	4.1.7-860.1
gawk	3.1.3-210.1
gdbm	1.8.3-228.1
glibc	2.3.3-98.31
glibc-locale	2.3.3-98.31
gpg	1.2.4-68.1
gpm	1.20.1-301.1
grep	2.5.1-427.1
groff	1.17.2-881.1
gzip	1.3.5-136.1
hdparm	5.5-41.3
heimdal-lib	0.6.1rc3-55.3
hfsutils	3.2.6-1038.1
hotplug	0.44-32.21
howtoenh	2004.4.4-0.4
hwinfo	8.61-0.3
info	4.6-61.1
insserv	1.00.2-85.1
iproute2	2.4.7-866.5
iputils	ss021109-147.1

All Platforms:	
kbd	1.12-26.1
ksymoops	2.4.9-135.1
laus	0.2-14.17
ldapcpplib	0.0.3-21.3
less	382-34.8
libacl	2.2.21-54.1
libattr	2.4.12-56.1
libgcc	3.3.3-43.24
libselinux	1.8-16.1
libstdc++	3.3.3-43.24
libxcrypt	2.1.90-61.3
libxml2	2.6.7-28.1
liby2util	2.9.25-0.2
logrotate	3.7-31.1
lprng	3.8.25-37.1
lsof	4.70-30.1
lukemftp	1.5-578.1
m4	1.4o-622.1
mailx	10.6-65.1
man	2.4.1-214.1
man-pages	1.67-1.6
mingetty	0.9.6s-73.1
mkinitrd	1.0-199.53
mktemp	1.5-729.1
module-init-tools	3.0_pre10-37.16
ncurses	5.4-61.4
net-tools	1.60-543.6
netcat	1.10-864.1
netcfg	9-17.1
openldap2-client	2.2.6-37.19
openslp	1.1.5-73.9
openssh	3.8p1-37.9
openssl	0.9.7d-15.13
pam-laus	0.77-4.3
pam-modules	9-18.5
parted	1.6.6-138.3
pciutils	2.1.11-192.8
pcre	4.4-109.1
perl	5.8.3-32.1
perl-Config-Crontab	1.03-46.1
permissions	2004.7.30-0.2
popt	1.7-176.7
postfix	2.1.1-1.4
procps	3.2.1-5.3
psmisc	21.4-39.1
pwdutils	2.6.4-2.16
readline	4.3-306.5

All Platforms:	
release-notes	9.1-8.40
resmgr	0.9.8-47.3
rpm	4.1.1-177.6
scsi	1.7_2.34_1.06_0.11-9.9
sed	4.0.9-31.1
sitar	0.8.11-17.1
sles-admin_en	9.1.0.5-0.16
sles-release	9-82.11
star	1.5a38-26.1
stunnel	4.05-20.1
submount	0.9-33.6
suse-build-key	1.0-662.4
sysconfig	0.31.0-15.29
syslogd	1.4.1-519.3
sysvinit	2.85-21.3
tar	1.13.25-325.3
tcpd	7.6-710.1
telnet	1.1-38.3
terminfo	5.4-61.4
texinfo	4.6-61.1
timezone	2.3.3-98.31
udev	021-36.32
usbutils	0.11-211.1
utempter	0.5.2-385.4
util-linux	2.12-72.20
vim	6.2-235.1
vsftpd	1.2.1-69.3
w3m	0.4.1_m17n_20030308-201.1
wget	1.9.1-45.3
xinetd	2.3.13-39.3
yast2	2.9.75-0.2
yast2-bootloader	2.9.34-0.3
yast2-core	2.9.94-1.2
yast2-country	2.9.24-0.2
yast2-inetd	2.9.12-21.1
yast2-installation	2.9.89-0.2
yast2-ldap	2.9.15-1.2
yast2-ldap-client	2.9.23-0.2
yast2-mail-aliases	2.9.16-0.2
yast2-mouse	2.9.11-4.1
yast2-ncurses	2.9.26-0.2
yast2-network	2.9.59-0.2
yast2-online-update	2.9.12-0.3
yast2-packagemanager	2.9.51-1.3
yast2-packager	2.9.51-0.2
yast2-pam	2.9.13-0.3
yast2-perl-bindings	2.9.34-1.2

All Platforms:	
yast2-runlevel	2.9.15-0.2
yast2-security	2.9.14-18.1
yast2-storage	2.9.58-0.3
yast2-sysconfig	2.9.15-0.2
yast2-theme-SuSELinux	2.9.13-0.4
yast2-trans-en_US	2.9.7-1.2
yast2-transfer	2.9.3-0.2
yast2-update	2.9.27-0.2
yast2-users	2.9.39-0.3
yast2-x11	2.9.11-0.2
yast2-xml	2.9.8-19.1
zlib	1.2.1-70.6
Only on i386	
grub	0.94-45.3
isapnp	1.26-489.1
kernel-default	2.6.5-7.108
kernel-smp	2.6.5-7.108
lilo	22.3.4-508.1
Only on x86_64	
bzip2-32bit	9-200407011229
cracklib-32bit	9-200407011229
cyrus-sasl-32bit	9-200407011229
db-32bit	9-200407011229
e2fsprogs-32bit	9-200407011229
file-32bit	9-200407011229
gdbm-32bit	9-200407011229
glibc-32bit	9-200407011233
glibc-locale-32bit	9-200407011229
grub	0.94-45.3
heimdal-lib-32bit	9-200407011229
irqbalance	0.09-37.1
isapnp	1.26-489.1
kernel-default	2.6.5-7.108
kernel-smp	2.6.5-7.108
laus-32bit	9-200407011229
libacl-32bit	9-200407011229
libattr-32bit	9-200407011229
libselinux-32bit	9-200407011229
libxcrypt-32bit	9-200407011229
libxml2-32bit	9-200407011229
lilo	22.3.4-508.1
ncurses-32bit	9-200407011229
openldap2-client-32bit	9-200407011229
openssl-32bit	9-200407011229
pcre-32bit	9-200407011229

Only on x86_64	
perl-32bit	9-200407011229
popt-32bit	9-200407011229
readline-32bit	9-200407011229
resmgr-32bit	9-200407011229
utempter-32bit	9-200407011229
zlib-32bit	9-200407011229
Only on s390x	
bzip2-32bit	9-200407011411
cracklib-32bit	9-200407011411
cyrus-sasl-32bit	9-200407011411
db-32bit	9-200407011411
e2fsprogs-32bit	9-200407011411
file-32bit	9-200407011411
gdbm-32bit	9-200407011411
glibc-32bit	9-200407011411
glibc-locale-32bit	9-200407011411
heimdal-lib-32bit	9-200407011411
kernel-s390x	2.6.5-7.108
laus-32bit	9-200407011411
libacl-32bit	9-200407011411
libattr-32bit	9-200407011411
libgcc-32bit	9-200407011411
libselinux-32bit	9-200407011411
libxcrypt-32bit	9-200407011411
libxml2-32bit	9-200407011411
ncurses-32bit	9-200407011411
openldap2-client-32bit	9-200407011411
openssl-32bit	9-200407011411
pcre-32bit	9-200407011411
perl-32bit	9-200407011411
popt-32bit	9-200407011411
readline-32bit	9-200407011411
resmgr-32bit	9-200407011411
s390-tools	1.3.1-0.3
utempter-32bit	9-200407011411
zlib-32bit	9-200408261522
Only on iSeries	
baselibs-64bit	9-200407011606
bzip2-64bit	9-200407011606
cracklib-64bit	9-200407011606
cyrus-sasl-64bit	9-200407011606
db-64bit	9-200407011606
e2fsprogs-64bit	9-200407011606
file-64bit	9-200407011606
gdbm-64bit	9-200407011606

Only on iSeries	
glibc-64bit	9-200407011606
glibc-locale-64bit	9-200407011606
heimdal-lib-64bit	9-200407011606
kernel-pseries64	2.6.5-7.108
laus-64bit	9-200407011606
libacl-64bit	9-200407011606
libattr-64bit	9-200407011606
libgcc-64bit	9-200407011606
libselinux-64bit	9-200407011606
libstdc++-64bit	9-200407011606
libxcrypt-64bit	9-200407011606
libxml2-64bit	9-200407011606
lilo	0.0.15-22.17
ncurses-64bit	9-200407011606
openldap2-client-64bit	9-200407011606
openssl-64bit	9-200407011606
pcre-64bit	9-200407011606
pdisk	0.8a-445.1
perl-64bit	9-200407011606
popt-64bit	9-200407011606
ppc64-utils	0.9-1.1
readline-64bit	9-200407011606
resmgr-64bit	9-200407011606
utempter-64bit	9-200407011606
zlib-64bit	9-200407011606
Only on pSeries	
bzip2-64bit	9-200407011606
cracklib-64bit	9-200407011606
cyrus-sasl-64bit	9-200407011606
db-64bit	9-200407011606
e2fsprogs-64bit	9-200407011606
file-64bit	9-200407011606
gdbm-64bit	9-200407011606
glibc-64bit	9-200407011606
heimdal-lib-64bit	9-200407011606
kernel-pseries64	2.6.5-7.108
laus-64bit	9-200407011606
libacl-64bit	9-200407011606
libattr-64bit	9-200407011606
libgcc-64bit	9-200407011606
libselinux-64bit	9-200407011606
libstdc++-64bit	9-200407011606
libxcrypt-64bit	9-200407011606
libxml2-64bit	9-200407011606
lilo	0.0.15-22.17
ncurses-64bit	9-200407011606

Only on pSeries	
openldap2-client-64bit	9-200407011606
openssl-64bit	9-200407011606
pcre-64bit	9-200407011606
pdisk	0.8a-445.1
perl-64bit	9-200407011606
popt-64bit	9-200407011606
ppc64-utils	0.9-1.1
readline-64bit	9-200407011606
resmgr-64bit	9-200407011606
utempter-64bit	9-200407011606
zlib-64bit	9-200407011606

Table 2-1. List of packages on TOE

3 Hardware architecture

The target of evaluation includes the IBM xSeries, pSeries, iSeries, zSeries, and eServer 325. This section describes the hardware architecture of these eServer systems. For more detailed information on Linux support and resources for the entire eServer line, please refer to the following IBM Redbook:

Linux Handbook, A guide to IBM Linux Solutions and Resources, Nick Harris et al.
http://www.redbooks.ibm.com/redbooks/pdfs/sg247000.pdf

3.1 xSeries

IBM xSeries 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.

xSeries servers are available in the following four categories:

- Universal Servers
- Rack Optimized Servers
- Extremely Scalable Servers
- BladeCenter™ Servers

The following provides a brief overview of xSeries hardware. For more detailed information on xSeries hardware, please refer to the xSeries hardware Web site:

http://www.pc.ibm.com/us/eserver/xseries

3.1.1 xSeries hardware overview

IBM xSeries 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 xSeries 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 xSeries servers, please refer to the corresponding data sheets at the following xSeries literature Web site:

http://www.pc.ibm.com/us/eserver/xseries/literature

The following tables list various hardware components, such as processors, memory, and storage devices, for xSeries servers.

xSeries Family

	Universal				Rack-optimized
xSeries model	**x205**	**x225**	**x235**	**x255**	**x305**
Form factor/height	Tower, Rack/4U	Tower, Rack/4U	Tower, Rack/5U	Tower, Rack/7U	Rack/1U
Intel processor (max)	2.4GHz Pentium 4	2.4GHz Xeon	2.4GHz Xeon	2.0GHz Xeon Processor MP	2.6GHz Pentium 4
Number of processors (std/max)	1/1	1/2	1/2	1/4	1/1
Cache (max)	512KB L2	512KB L2	512KB L2	2MB L3	512KB L2
Memory (max)	2GB PC2100	8GB PC2100 DDR Chipkill	12GB PC2100 DDR Chipkill	12GB PC1600 DDR Chipkill	4GB PC2100
Expansion slots (total/ Active PCI)	5/0	5/0	6/2 Active PCI-X	7/6 Active PCI-X	2/0 PCI-X
Disk bays (total/hot-swap)	4 non-hot-swap SCSI, 3 hot-swap SCSI, 3 IDE	6/6, 6 hot-swap SCSI, 4 non-hot-swap SCSI	9/9 with optional 3-pack hot-swap HDD kit	12/12 with optional 6-pack hot-swap HDD kit	2/0
CD-ROM/ diskette drive	Yes	Yes	Yes	Yes	Yes
Maximum internal storage	360GB2 IDE 587.2GB SCSI	880.8GB hot-swap, 587.2GB non-hot-swap	1.3TB	1.76TB	240GB IDE or 293.6GB SCSI
Network	Integrated 10/100/1000 Ethernet	Integrated 10/100/1000 Ethernet	Integrated 10/100/1000 Ethernet	Integrated 10/100/1000 Ethernet	Dual integrated 10/100/1000 Ethernet
System Management processor	Supports optional Remote Supervisor Adapter	Supports optional Remote Supervisor Adapter	Integrated (supports Remote Supervisor Adapter)	Integrated (supports Remote Supervisor Adapter)	Supports optional Remote Supervisor Adapter
Power supply (std/max)	340W 1/1	425W or (2) 350W hot-swap	560W 1 or 2/2 hot-swap	370W 2/4 hot-swap	220W 1/1
Hot-swap components	HDDs (select models)	Power supply, HDDs (select models)	Power supply, fans, HDDs, PCI-X	Power supply, fans, HDDs, PCI-X	N/A
Light Path Diagnostics	Limited	Limited	Yes	Yes	N/A
RAID support	Optional	Integrated RAID-1	Integrated RAID-1	Optional	Optional

xSeries Family

	Rack-optimized			Scalable	
xSeries model	**x335**	**x345**	**x360**	**x440 Entry**	**x440**
Form factor/height	Rack/1U	Rack/2U	Rack/3U	Rack/4U	Rack/4U
Intel processor (max)	2.8GHz Xeon	2.4GHz Xeon	2.0GHz Xeon Processor MP	2.4GHz Xeon	2.0GHz Xeon Processor MP
Number of processors (std/max)	1/2	1/2	1/4	2/4 or 4/4	2/8 or 4/8
Cache (max)	512KB L2	512KB L2	2MB L3	512KB L2 32MB L4/2-way	2MB L3 32MB L4/4-way
Memory (max)	4GB PC2100 DDR Chipkill	8GB PC2100 DDR Chipkill	8GB PC1600 DDR Chipkill	64GB (32GB/2-way or 64GB/4-way) Chipkill	64GB (32GB/4-way or 64GB/8-way) Chipkill
Expansion slots (total/ Active PCI)	2/0 PCI-X	4/0 PCI-X 1/0 PCI	6/6 Active PCI-X	6/6 Active PCI-X, optional remote I/O	6/6 Active PCI-X, optional remote I/O
Disk bays (total/hot-swap)	2/2	6/6	3/3	2/2	2/2
CD-ROM/ diskette drive	Yes	Yes	Yes	Yes	Yes
Maximum internal storage	240GB IDE or 293.6GB SCSI	880.8GB	220.2GB	146.8GB	146.8GB
Network	Dual integrated 10/100/1000 Ethernet	Dual integrated 10/100/1000 Ethernet	Integrated 10/100 Ethernet	Integrated 10/100/1000 Ethernet	Integrated 10/100/1000 Ethernet
System Management processor	Integrated (supports Remote Supervisor Adapter)	Integrated (supports Remote Supervisor Adapter)	Remote Supervisor Adapter in dedicated slot	Remote Supervisor Adapter in dedicated slot	Remote Supervisor Adapter in dedicated slot
Power supply (std/max)	332W 1/1	350W 1/2 hot-swap	370W 1 or 2/3 hot-swap	1050W 2/2 hot-swap	1050W 2/2 hot-swap
Hot-swap components	HDDs	Power supply, fans, HDDs	Power supply, fans, HDDs	Power supply, fans, HDDs, PCI-X	Power supply, fans, HDDs, PCI-X
Light Path Diagnostics	Yes	Yes	Yes	Yes	Yes
RAID support	Integrated RAID-1	Integrated RAID-1	Optional	Optional	Optional

IBM @server BladeCenter Family

	BladeCenter		BladeCenter HS20
Form factor/height	Rack/7U	Processor	Intel Xeon processor starting at 2.0GHz
Blade server bays	14	Number of processors (std/max)	1/2
Standard media	CD-ROM and diskette drive accessible from each blade server	Level 2 cache	512KB
Switch modules	4 module bays for FC or Gigabit Ethernet switches	Front side bus	Starting at 400MHz
Power supply module	Up to 4 hot-swap 1200W with load-balancing and failover capabilities	Memory	Up to 8GB PC2100 DDR Chipkill
Cooling modules	2 hot-swap and redundant blowers standard	Internal hard disk drives	Up to 2 IDE (and up to 2 hot-swap Ultra320 SCSI drives with optional SCSI storage expansion unit)
Systems management hardware	Up to 2 management modules	Maximum internal storage	80GB IDE/146.8GB SCSI
I/O ports	Keyboard, video, mouse, Ethernet, USB	RAID support	Integrated mirroring (with optional SCSI storage expansion unit)
		Network	Dual integrated 10/100/1000 Ethernet
		I/O upgrade	1 expansion card connection
		Systems management hardware	Integrated systems management processor
		Systems management software	IBM Director with systems management and deployment tools
		Predictive Failure Analysis	Hard disk drives, processors, blowers, memory
		Light Path Diagnostics	Blade server, processor, memory, power supplies, blowers, switch module, management module, hard disk drives and expansion card
		Limited warranty and support	3-year on-site limited warranty
		External storage	Support for IBM TotalStorage solutions (including FAStT and NAS family of products)
		Operating systems supported	Microsoft Windows® 2000 Server/ Advanced Server, Red Hat Linux, SUSE Linux, Novell Netware

Figure 3-1. IBM xSeries models

3.1.2 xSeries hardware architecture

IBM xSeries servers are powered by Intel Pentium® III, Pentium 4, and Xeon, processors. For detailed specification information for each of these processors, please refer to the Intel processor spec-finder Web site at http://*processorfinder.intel.com/scripts/default.asp*

For architectural details on all xSeries models, and for detailed information on individual components such as memory, cache, and chipset, please refer to the IBM xSeries Technical Principles Training (XTW01) at *http://www-1.ibm.com/servers/eserver/education/xseries/technical.html*

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

3.2 pSeries

IBM pSeries systems are PowerPC processor-based systems that provide high availability, scalability, and powerful 64-bit computing performance. IBM pSeries offers a broad range of proven systems with innovative price-performance features.

The pSeries servers are available in the following three categories:

- Entry level servers
- Mid-range servers
- High-end servers

The following provides a brief overview of pSeries hardware. For more detailed information on pSeries hardware, please refer to the following pSeries hardware Web site:

http://www-1.ibm.com/servers/eserver/pseries

3.2.1 pSeries hardware overview

IBM pSeries 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 pSeries 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 pSeries servers, please refer to the corresponding data sheets at the pSeries literature Web site:

http://www-1.ibm.com/servers/eserver/pseries/literature

The following lists various POWER5 based pSeries models and their features:

@server p5 520 rack system with I/O drawer

p5-520 at a glance

Available configurations	
Microprocessors	2-way 64-bit 1.65 GHz POWER5 processors
Level 2 (L2) cache	1.9MB
Level 3 (L3) cache	36MB
RAM (memory)	1GB – 32GB ECC DDR1 SDRAM
Internal storage	8.2TB (with optional I/O drawers)
Processor-to-memory bandwidth	12.8GB/second
L2 to L3 cache bandwidth	26.4GB/second
RIO-2 I/O subsystem bandwidth	4.4GB/second
Internal disk bays	Four standard plus four optional (36.4/73.4/146.8GB 10K rpm or 36.4GB/73.4GB 15K rpm disks)
Media bays	Two slimline and one standard
Adapter slots	Six hot-plug 3.3v PCI-X (2 – 32-bit/66 MHz; 4 - 64-bit/133 MHz)

@server p5 550 deskside system

p5-550 at a glance

Available configurations	
Microprocessors	2- or 4-way 64-bit 1.65 GHz POWER5 processors
Level 2 (L2) cache	1.9MB per processor card
Level 3 (L3) cache	36MB per processor card
RAM (memory)	1GB – 64GB
Internal storage	15.2TB (with optional I/O drawers)
Processor-to-memory bandwidth	25.5GB/sec for 4-way configurations
L2-to-L3 cache bandwidth	52.8GB/sec for 4-way configurations
RIO-2 I/O subsystem bandwidth	8.8GB/second
Internal disk bays	Four standard plus four optional (36.4/73.4/146.8GB 10K rpm or 36.4GB/73.4GB 15K rpm disks)
Media bays	Two slimline and one standard
Adapter slots	Five hot-plug 64-bit PCI-X (four long and one short), 3.3 volt

IBM @server p5 570 in rack con I/O drawer

p5-570 at a glance

Available configurations	Per module	p5-570 Express	p5-570 with 1.65 or 1.9 GHz processors
Microprocessors	2 or 4 64-bit 1.5 GHz, 1.65 GHz or 1.9 GHz POWER5 in the first module; 4 processors in all other modules	2, 4 or 8 64-bit 1.5 GHz POWER5	2, 4, 8, 12 or 16 64-bit 1.65 or 1.9 GHz POWER5
Level 3 (L3) cache (maximum)	36MB (2 processor module) or 72MB (4 processor module)	144MB	288MB
Shared system memory (minimum/ maximum)	2GB/128GB[1] 2GB/16GB[2]	2GB/256GB[1]	2GB/512GB[1] 2GB/64GB[2]
Processor-to-memory bandwidth (maximum)	25.5GB/sec.[1] 51.1GB/sec.[2]	51.0GB/sec.[1]	102.1GB/sec.[1] 204.6GB/sec.[2]
L2-to-L3 cache bandwidth (maximum)	48.0GB/sec.[1] 60.8GB/sec.[2]	96.0GB/sec.[1]	192.0GB/sec.[1] 243.2GB/sec.[2]
Internal disk bays (maximum)	6 on a split backplane (3+3)	12 (2 split backplanes)	24 (4 split backplanes)
Media bays (maximum)	Two hot-plug slimline media bays	Four hot-plug slimline media bays	Eight hot-plug slimline media bays
Adapter slots (PCI-X)	Six hot-plug blind-swap: Five long 64-bit 133 MHz 3.3v; One short 64-bit 133 MHz 3.3v	12 hot-plug blind-swap: 10 long 64-bit 133 MHz 3.3v; Two short 64-bit 133 MHz 3.3v	24 hot-plug blind-swap: 20 long 64-bit 133 MHz 3.3v; Four short 64-bit 133 MHz 3.3v

For additional hardware information on POWER5 based pSeries systems, please refer to the following document:

http://www-1.ibm.com/servers/eserver/pseries/hardware/factsfeatures.pdf

The following figure displays various POWER4+ based pSeries models and their computational capacity.

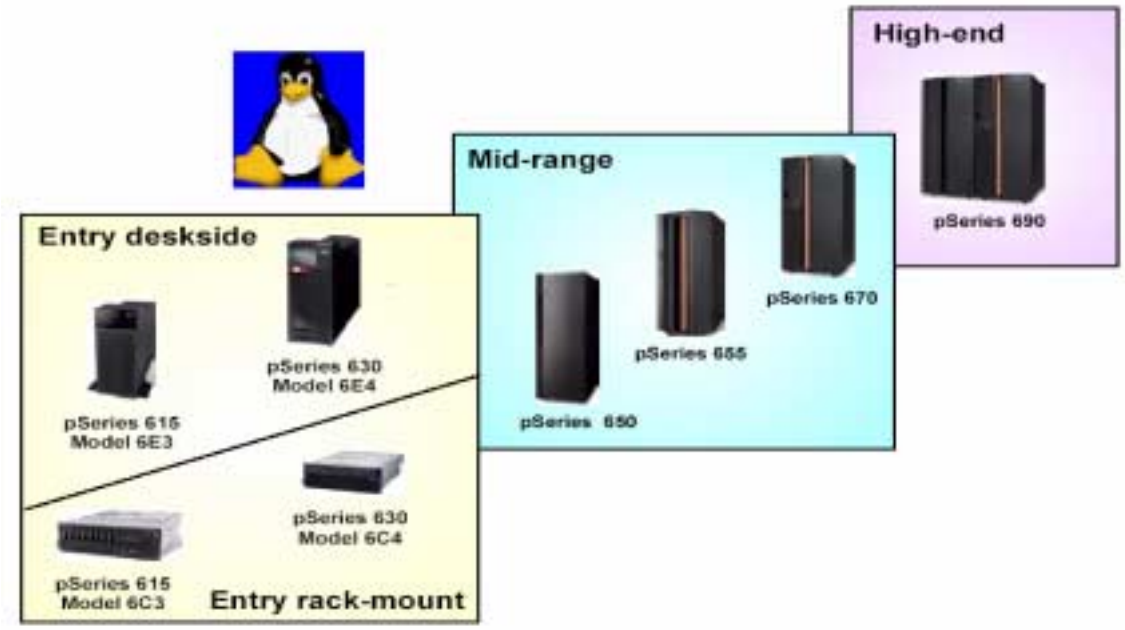

Figure 3-2. IBM pSeries models

For a detailed look at various peripherals such as storage devices, communications interfaces, storage interfaces, and display devices supported on these pSeries models, please refer to the following hardware matrix:

IBM eServer pSeries and IBM RS6000 Linux Facts and Features
http://www.ibm.com/servers/eserver/pseries/hardware/linux_facts.pdf

3.2.2 pSeries hardware architecture

IBM pSeries servers are powered by POWER3™, RS64-III, RS64 IV, POWER4™, POWER4+, and POWER5™ processors. For detailed specification information for each of these processors, please refer to the PowerPC processor documentation at *http://www-3.ibm.com/chips/products/powerpc*

For information on POWER5 architecture, please refer to *http://www-1.ibm.com/technology/power/?ca=power&met=web&me=powercallout*

For architectural details on all pSeries models, and for detailed information on individual components such as memory, cache, and chipset, please refer to the IBM pSeries technical documentation at the following Web sites:

http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base/hardware.htm
http://www-1.ibm.com/servers/eserver/pseries/library

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

3.3 iSeries

IBM iSeries systems are PowerPC, POWER4 and POWER5 processor-based servers that provide scalability, quick application integration, and server consolidation via industry leading logical partitioning. This logical partitioning supports multiple images of SLES, OS/400®, and i5/OS, and is available across

the iSeries product line. SLES on iSeries requires very little dedicated hardware because it can share RAID adapters, CD-ROM, and LAN adapters with OS/400 and i5/OS.

The following provides a brief overview of iSeries hardware. For more detailed information on iSeries hardware, please refer to the following iSeries hardware Web site:

http://www-1.ibm.com/servers/eserver/iseries

For information on POWER5 architecture, please refer to the following Web site:

http://www-1.ibm.com/technology/power/?ca=power&met=web&me=powercallout

For information on POWER5 processor based systems, please refer to the following iSeries i5 Redbook:

http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/ga195486.html?Open

For information on POWER5 processor's Symultaneous Multi Threading technology, please refer to the following Web site:

http://www-1.ibm.com/servers/eserver/iseries/perfmgmt/pdf/SMT.pdf

iSeries hardware overview

The IBM iSeries systems offer support for gigabytes of memory, large RAID configurations of SCSI and Fiber Channel disks, and options for high-speed networking. IBM iSeries 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. The following table lists different POWER4 processor-based iSeries models and the number of SLES partitions supported on them.

Series Model Support

270	# of CPUs	LPAR Shared CPU	Linux	Linux Max # of Partitions
2431	1	Yes	Yes	9
2432	1	Yes	Yes	9
2434	2	Yes	Yes	19
2452*	1	Yes	Yes	9
2454*	2	Yes	Yes	19
820	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
0150	1	Yes	Yes	9
0151	2	Yes	Yes	19
0152	4	Yes	Yes	31
2435	1	Yes	Yes	9
2436	1	Yes	Yes	9
2437	2	Yes	Yes	19
2438	4	Yes	Yes	31
2456*	1	Yes	Yes	9
2457*	2	Yes	Yes	19
2458*	4	Yes	Yes	31
2395	1	Yes	No	0
2396	1	Yes	No	0
2397	2	Yes*	Yes	1
2398	4	Yes*	Yes	3
2425*	1	Yes	No	0
2426*	2	Yes*	Yes	1
2427*	4	Yes*	Yes	3
830	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
2400	2	Yes*	Yes	1
2402	4	Yes*	Yes	3
2403	8	Yes*	Yes	7
0153	8	Yes*	Yes	7
2349	4/8	Yes*	Yes	3/7
840	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
2461	24	Yes	Yes	31
2352	8/12	Yes	Yes	31
2353	12/18	Yes	Yes	31
2354	18/24	Yes	Yes	31
0158	12	Yes	Yes	31
0159	24	Yes	Yes	31
2418	12	Yes*	Yes	11
2420	24	Yes*	Yes	23
2416	8/12	Yes*	Yes	11
2417	12/18	Yes*	Yes	17
2419	18/24	Yes*	Yes	23

iSeries Model Support

800	# of CPUs	LPAR Shared CPU	Linux	Linux Max # of Partitions
	1	Yes	Yes	9
810	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
	1	Yes	Yes	9
	2	Yes	Yes	19
825	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
	3	Yes	Yes	29
	4	Yes	Yes	31
	5	Yes	Yes	31
	6	Yes	Yes	31
870	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
	8/16	Yes	Yes	31
890	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
	16/24	Yes	Yes	31
	24/32	Yes	Yes	31
890	**# of CPUs**	**LPAR Shared CPU**	**Linux**	**Linux Max # of Partitions**
0197	24	Yes	Yes	31
0198	32	Yes	Yes	31
2487	16/8	Yes	Yes	31
2488	24/8	Yes	Yes	31

Resources

- Web site: http://www.ibm.com/eserver/iseries/linux
  White Paper, Linux Solutions Guide, Spec Sheet
  Redbooks; Implementation and Integration

Figure 3-3. IBM iSeries models

Figure 3-4. IBM iSeries POWER5 based i5 520

i5 520 Hardware Configurations

- Includes i5/OS and can add Windows, Linux, and AIX 5L operating systems
- 1-way and 2-way offerings provide from 500 CPW to 6,000 CPW
- Up to 32 GB memory
- Up to 278 disk drives - 19TB of capacity
- Up to 6 I/O expansion towers/drawers via High Speed Link
- Up to 90 PCI-X slots, 192 WAN lines, 36 LANs
- Integrated DVD-ROM or DVD-RAM
- Base 2-line WAN and base IOP
- Up to 18 Integrated xSeries Servers
- Redundant, hot-plug components for additional reliability
- Choice of rack-mounted or Deskside tower

Software	**i5/OS V5R3**
Memory (Range)	**Up to 32GB**
Disk (Range)	**Up to 19TB**
Processor performance	**500 - 6000 CPW**

The following provides brief description of the POWER5 processor based iSeries model i5 570.

Figure 3-5. IBM iSeries POWER5 based i5 570

i5 570 Hardware Configurations

- Includes i5/OS and can add Windows, Linux, and AIX 5L operating systems

- 1/2-way and 2/4-way offerings provide from 3,200 CPW to 11,200 CPW
- 2 GB to 64 GB memory
- 1 to 546 disk drives - 39TB of capacity
- Up to 12 I/O expansion towers/drawers via High Speed Link
- Up to 173 PCI-X slots, 320 WAN lines, 128 LANs
- Integrated Ethernet LAN and disk controllers
- Integrated DVD-ROM or DVD-RAM
- Base 2-line WAN and base IOP
- Up to 36 Integrated xSeries Servers
- Up to 16 Integrated xSeries Adapters
- Redundant, hot-plug components for additional reliability
- Rack-optimized design

Software	**i5/OS V5R3**
Memory (Range)	**Up to 64GB**
Disk (Range)	**Up to 39TB**
Processor performance	**3300 - 11700 CPW**

3.3.1 iSeries hardware architecture

IBM iSeries servers are powered by POWER3, RS64-III, RS64 IV, POWER4, POWER4+, and POWER5 and processors. For detailed specification information for each of these processors, please refer to the PowerPC processor documentation at *http://www-3.ibm.com/chips/products/powerpc*

iSeries systems differ from pSeries systems in their I/O architecture. The Linux environment on iSeries supports two types of I/O: virtual and direct.

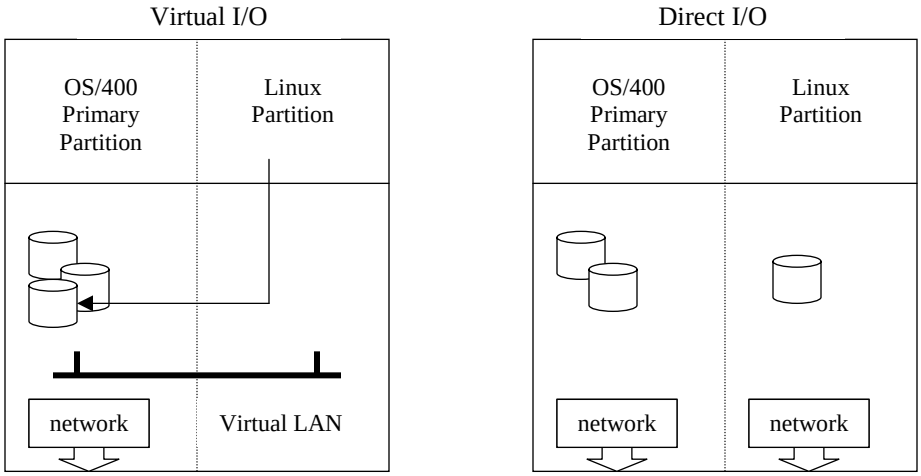

Figure 3-6. I/O architectures of iSeries

With virtual I/O, the I/O resources such as disk drive, tape drive, and CD-ROM, are owned by an OS/400 or an i5/OS partition. OS/400 or i5/OS shares the resources with SLES. The shared resources are under OS/400 or i5/OS management. For example, OS/400 or i5/OS provides the RAID protection and some backup/restore facilities for the Linux environment. Virtual Ethernet provides sixteen 1 GB Ethernet communications paths between partitions. With direct I/O, the I/O resources are owned by the partition running SLES. Disk drives, LAN or WAN adapters connected with direct I/O are under the control of SLES.

For architectural details on all iSeries models, and for detailed information on individual components, such as memory, cache, and chipset, please refer to the following IBM iSeries technical documentation:

IBM iSeries Hardware, ITSO Technical Overview
ftp://ftp.software.ibm.com/as400/marketing/pdf/v5r1/hw.pdf
http://www-1.ibm.com/servers/eserver/iseries/library

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

3.4 zSeries

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

The following provides a brief overview of zSeries hardware. For more detailed information on zSeries hardware, please refer to the following zSeries hardware Web site:

http://www-1.ibm.com/servers/eserver/zseries

3.4.1 zSeries hardware overview

zSeries hardware runs z/Architecture™ and S/390® Enterprise Server Architecture (ESA). IBM zSeries 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. The following gives an overview of the zSeries hardware models. For more detailed information on specific models, please refer to the following model-specific Web sites:

http://www-1.ibm.com/servers/s390/pes/
http://www-1.ibm.com/servers/eserver/zseries/800.html
http://www-1.ibm.com/servers/eserver/zseries/900.html
http://www-1.ibm.com/servers/eserver/zseries/990.html

The following table lists S/390 hardware.

S/390 G5/G6 enterprise servers at a glance							
Hardware models and number of processors	9672 G5	*IBM S/390 Architecture*					
		RA6 (1-way)	R16 (1-way)	RB6 (2-way)	R26 (2-way)	RC6 (3-way)	RD6 (4-way)
		R36 (3-way)	R46 (4-way)	R56 (5-way)	R66 (6-way)	R76 (7-way)	R86 (8-way)
		R96 (9-way)	RX6 (10-way)	Y36 (3-way)	Y46 (4-way)		
		Y56 (5-way)	Y66 (6-way)	Y76 (7-way)	Y86 (8-way)		
		Y96 (9-way)	YX6 (10-way)				
	9672 G6	X17 (1-way)	X27 (2-way)	X37 (3-way)	X47 (4-way)	X57 (5-way)	X67 (6-way)
		X77 (7-way)	X87 (8-way)	X97 (9-way)	XX7 (10-way)	XY7 (11-way)	XZ7 (12-way)
		Z17 (1-way)	Z27 (2-way)	Z37 (3-way)	Z47 (4-way)	Z57 (5-way)	Z67 (6-way)
		Z77 (7-way)	Z87 (8-way)	Z97 (9-way)	ZX7 (10-way)	ZY7 (11-way)	ZZ7 (12-way)
Coupling facility	9672 Model R06	1 – 10 ICFs					
	ICF	Up to 3/7/9 (RA6 - RD6 / R36 - Y96 / X17 - XY7 and Z17 - ZY7)					
Channels	Minimum	3 or 4/4/0/3 (Parallel/ESCON/FICON/Total)					
	Maximum	96/256/36*/256 (Parallel/ESCON/FICON/Total)					
	Increments	3 or 4/4/1 (Parallel/ESCON/FICON)					
	General	Channel maximums can be decreased by the use of OSAs and/or coupling links					
Cryptographic	— CMOS Cryptographic Coprocessor	2 - standard					
	— PCI Cryptographic Coprocessor	up to 8 - optional					
Processor storage	Minimum	1 GB					
	Maximum	Up to 32 GB					
	General	Central and expanded storage are user-definable					
	Exceptions	RA6, RB6, R26, RC6, RD6 models maximum 12 GB R36 - RX6 models minimum 2 GB Y76, Y86, Y96, YX6 models minimum 8 GB Y36-Y66, Xn7, Zn7 models minimum 5 GB					
G5 upgradability	Upgradable from R3, G3 and G4 models Upgradable within G5 family						
G6 upgradability	Upgradable from G3, G4 and G5 models Upgradable within G6 family						
R06 upgradability	Upgradable from 9674 C04 and C05 models Upgradable within R06 models Upgradable to G5 RA6-YX6 models Upgradable to G6 X37-ZZ7 models						
Maximum coupling connections	HiPerLinks	32 (Any G5 or G6)					
	IC	32 (RA6 - ZZ7) (Not on R06)					
	ICB	6/18/24/18 (RA6 - RD6/R36 - YX6/R06/X17 - ZZ7)					
Physical configuration	1 Frame	Minimum	Weight (unpacked): 612 kg (1346 lbs) Footprint: 1.0 Sq meters (10.4 Sq feet) Service clearance 2.5 Sq meters (27.4 Sq feet) Input power: 0.6 kVA Heat output: 2.0 KBTU/hr				
	2 Frame	Maximum	Weight (unpacked): 938 kg (2057 lbs) Footprint: 1.8 Sq meters (19.7 Sq feet) Service clearance: 4.8 q meters (51.9 Sq feet) Input power: 5.5 kVA Heat output: 18.8 KBTU/hr				
General	Conforms to EIA guidelines for frames Employs standard 24-inch cage enclosures						

Table 3-1. IBM S/390 hardware

The following table lists zSeries 800 hardware.

IBM @server zSeries 800 enterprise server at a glance	
Hardware Models	
General Purpose Models	0A1, 0B1, 0C1, 0E1, 0X2, 001, 0A2, 002, 003, 004
Coupling Facility Model	0CF (1–4 Internal Coupling Facilities)
Dedicated Linux Model	0LF (1–4 Integrated Facilities for Linux)
Channels	
Minimum	0/0/0/0 (ESCON/FICON Express/ OSA-Express/HiperSockets)
Maximum	240/32/24/4 (ESCON/FICON Express/OSA-Express/HiperSockets)
Increments	4/2/2/1 (ESCON/FICON Express/OSA-Express/HiperSockets)
Cryptographic Coprocessor	
PCI Crypto Coprocessor	up to 16 optional (up to 8 cards)
PCI Crypto Accelerator	up to 12 optional (up to 6 cards)
CMOS Coprocessor	optional with increments of 0 to 2
Processor Memory	**All Models**
Minimum	8 GB
Maximum	32 GB
Increments	8 GB
Upgradeability	Upgradeable within zSeries 800 and from a zSeries 800 Model 004 to a zSeries 900 Model 104
Physical Configuration	**All Models**
Single Frame	
Weight (unpacked)	545 Kg (1201 lbs.)
Footprint	0.83 Sq. meters (8.9 Sq. feet)
Service Clearance	5.99 Sq. meters (64.5 Sq. feet) all sides require clearance of 762 mm (30 inches)
Input Power	200 - 240 V single phase
Heat Output	10.0 kBTU/hr
Air Flow	11.1 cubic m/mm (400 cubic ft/min)
Height	1810 mm (71.3 inches)
Software	• z/OS basic and LPAR mode: z/OS V1.1 and subsequent releases • z/OS.e™ V1.3 and subsequent releases in LPAR mode only • z/VM basic and LPAR mode: z/VM V3.1, z/VM V4.1 and subsequent releases • Linux for zSeries basic and LPAR mode: Redhat, SuSE, Turbolinux • OS/390® basic and LPAR mode: OS/390 V2.9 and subsequent releases • VM basic and LPAR mode: VM/ESA® V2.4 • VSE basic and LPAR mode: VSE/ESA™ V2.5 and subsequent releases • TPF V4.1 (ESA mode only) • Linux for S/390 basic and LPAR mode: Redhat, SuSE, Turbolinux

Table 3-2. IBM zSeries 800 hardware

The following table lists zSeries 900 hardware.

IBM @server zSeries 900 enterprise server at a glance						
Hardware models						
General Purpose Models	101-109, 110-116, 210-216					
Capacity Models	1C1-1C9, 2C1-2C9					
Coupling Facility	Model 100 Coupling Facility: 1-9 Internal Coupling Facilities					
Coupling Links	**Links**	**IC**	**ICB-3**	**ICB**	**ISC-3**	**Max # Links**
	z900 - 100	32	16	16	32, 42, w/RPQ	64
	z900 server	32	16	8,12/16 w/RPQ	32	32
Channels						
Minimum	0[1]/0/0/0/0/0 (Parallel/ESCON®/FICON™/FICON Express/OSA-Express/HiperSockets)					
Maximum	88[1]/256/96/96/24/4 (Parallel/ESCON/FICON/FICON Express/OSA-Express/HiperSockets)					
Increments	4[1]/4/2/2/2/1 (Parallel/ESCON/FICON/FICON Express/OSA-Express/HiperSockets)					
Cryptographic	CMOS Cryptographic Coprocessor — 2 standard PCI Crypto Coprocessor — up to 16 optional (up to 8 cards) PCI Crypto Accelerator — up to 12 optional (up to 6 cards)					
Processor memory	**Model 101-109**		**1C1-1C9/2C1-2C9**		**110-116/210-216**	
Minimum	5 GB		10 GB		10 GB	
Maximum	32 GB		64 GB		64 GB	
Upgradeability	Upgradeable from G5/G6, R06, IBM @server zSeries 800 (z800) models Upgradeable within zSeries 900. Upgradeable to z990 (except z900-100)					
Physical configuration	**Models 101-116 and 1C1-1C9,**		**2C1-2C9 and 210-216**			
	1 frame, minimum[2]		**2 frame, maximum[3]**			
Weight (unpacked)	917 kg (2021 lbs)		1866 kg (4113 lbs)			
Footprint	1.32 Sq meters		2.81 Sq meters (14.2 Sq feet)		(30.3 Sq feet)	
Service clearance	3.04 Sq meters		6.18 Sq meters (32.7 Sq feet)		(66.5 Sq feet)	
Input power	5.3 kVA		12.4 kVA			
Heat output	18.1 KBTU/hr		42.1 KBTU/hr			
Air Flow	CFM 800, m^3/m in 22.2		CFM 2223, m^3/m in 61.7			
Height	200.4 cm (79.8 inches)		200.4 cm (79.8 inches)			
General	Conforms to EIA guidelines for frames Employs standard 24-inch cage enclosures					
Software	z/OS basic and LPAR mode:		z/OS 1.1 and subsequent releases			
	z/VM basic and LPAR mode:		z/VM 3.1 and subsequent releases			
	Linux for zSeries basic and LPAR mode:		Red Hat, SuSE, Turbolinux			
	OS/390® basic and LPAR mode:		OS/390 2.8 and subsequent releases			
	VM basic and LPAR mode:		VM/ESA® 2.4 and subsequent releases			
	VSE basic and LPAR mode:		VSE/ESA™ 2.4 and subsequent releases			
	TPF:		TPF V4R1 (ESA mode only)			
	Linux for S/390 basic and LPAR mode:		Red Hat, SuSE, Turbolinux			

Table 3-3. IBM zSeries 900 hardware

The following table lists zSeries 990 hardware.

IBM @server zSeries 990 enterprise server at a glance

Hardware Models							
General Purpose / Coupling Facility / Linux	A08, B16, C24*, D32*						
Coupling Links	Links	ISC-3	IC	ICB-2	ICB-3	ICB-4	Max # Links
	z990	32	32	8	16	16	64[1]
Channels							
Minimum	0/0/0/0 (ESCON®/FICON Express™/OSA-Express/HiperSockets™)						
Maximum	512/120/48/16 (ESCON/FICON Express/OSA-Express/HiperSockets)						
Increments	4/2/2/1 (ESCON/FICON Express/OSA-Express/HiperSockets)						
Cryptographic	PCIX Crypto Coprocessor* — up to 4 optional (up to 4 cards)						
	PCI Crypto Accelerator — up to 12 optional (up to 6 cards)						
Processor Memory	**Model A08**	**B16**	**C24**	**D32**			
Minimum	8 GB	8 GB	8 GB	8 GB			
Maximum	64 GB	128 GB	192 GB	256 GB			
Upgradeability	Upgradeable within zSeries 990						
	Upgrade from zSeries 900						
Physical Configuration	**Models A08, 2 frame, minimum[2]**			**Models D32, 2 frame, maximum[3]**			
Weight (unpacked)	1174 kg			2007 kg			
Footprint	2.49 Sq meters			2.49 Sq. meters			
Service Clearance	5.45 Sq meters			5.45 Sq. meters			
Input Power	6.74 kVA			21.39 kVA			
Heat Output	22.92 KBTU/hr			72.73 KBTU/hr			
Air Flow	CFM 1450, m^3/m			CFM 3250, m^3/m			
Height	194.1 cm (76.4 inches)			194.1 cm (76.4 inches)			
General	Conforms to EIA guidelines for frames						
Software	z/OS® LPAR mode:			z/OS 1.2 and subsequent releases			
	z/VM® LPAR mode:			z/VM 3.1, z/VM 4.2 and subsequent releases			
	Linux for zSeries LPAR mode:			Red Hat, SuSE, Turbolinux			
	OS/390® LPAR mode:			OS/390 2.10			
	VSE LPAR mode:			VSE/ESATM 2.5 and subsequent releases			
	TPF LPAR mode:			TPF 4.1 (ESA mode only)			
	Linux for S/390 LPAR mode:			Red Hat, SuSE, Turbolinux			

[1] 32 External and 32 Internal [2] Model A08 with one I/O Cage and no IBF [3] Model D32 with three I/O Cages and IBF

* Planned availability - October 31, 2003

Table 3-4. IBM zSeries 990 hardware

3.4.2 zSeries hardware architecture

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

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 zSeries system can be logically partitioned into a maximum of 30 separate Logical Partitions (LPAR). A single zSeries 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. Linux 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 zSeries system. SLES runs on zSeries in the z/VM guest mode. Virtualization of devices in the z/VM guest mode, allows SLES to operate with generic devices. z/VM maps these generic devices to actual devices.

The following diagram from the *Linux Handbook*[LH] illustrates z/VM concepts.

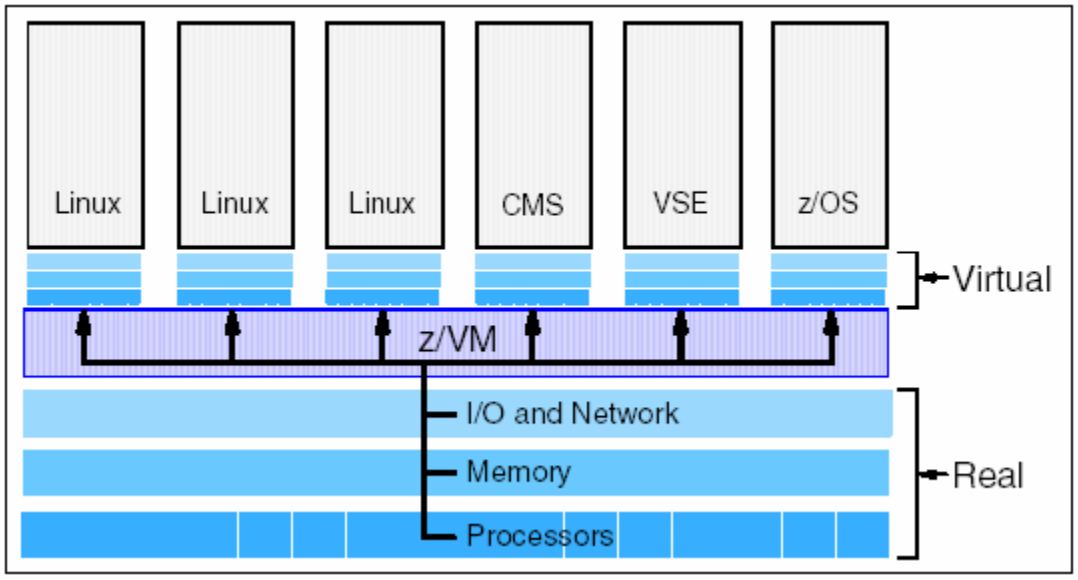

Figure 3-7. z/VM as hypervisor

For more detail on z/Architecture, please refer to the following z/Architecture document:

z/Architecture Principles of Operation
http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf

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

3.5 eServer 325

IBM eServer 325 systems are AMD Opteron processor-based systems that provide outstanding value for high performance computing in both 32-bit and 64-bit environments. The eServer 325 significantly improves on existing 32-bit applications and excels at 64-bit computing in performance, allowing for easy migration to 64-bit computing.

The following provides a brief overview of the eServer 325 hardware. For more detailed information on eServer 325 hardware, please refer to the following eServer 325 hardware Web site:

http://www.pc.ibm.com/us/eserver/opteron

3.5.1 eServer 325 hardware overview

The IBM eServer 325 systems offer support for up to two AMD Opteron processors, up to twelve gigabytes of memory, hot-swap SCSI or IDE disk drives, RAID-1 mirroring, and options for high-speed networking. IBM eServer 325 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. The following table lists the different eServer 325 hardware components.

e325 at a glance	
Form Factor	Rack/1U
Processor	AMD Opteron™ 240, 242, 246
Number of processors (std/max)	1/2
Cache (max)	1MB
Memory (std/max)	2 x 512MB/12GB[3] PC2700 ECC DDR SDRAM
Expansion Slots	2 (64-bit/100MHz)
Disk bays (total/hot-swap)	2/2
Maximum internal storage	293.6GB[3] Ultra320 SCSI/240GB IDE
Network	Dual integrated 10/100/1000 Ethernet
Power supply (std/max)	411W 1/1
Hot-swap components	Hard disk drives
RAID support	Integrated RAID-1 standard (SCSI model only)
Systems management	Integrated System Management Processor, Cluster Systems Management
Operating systems supported	SuSE Linux Enterprise Server 8, SUSE Linux 8.2 Professional
Limited warranty[4]	1-year onsite limited warranty

Table 3-5. IBM eServer 325 hardware

3.5.2 eServer 325 hardware architecture

IBM eServer 325 systems are powered by the AMD Opteron processor. For detailed specification information on the Opteron processor, please refer to the following processor documentation:

http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_9003,00.html

Opteron is based on AMD x86-64 architecture. AMD x86-64 architecture is an extension of x86 architecture to extend full support for 16-bit, 32-bit, and 64-bit applications that are running concurrently. The x86-64 architecture adds a new mode called *long mode*. Long mode is activated by a global control bit called LMA (Long Mode Active). 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: 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 8 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 be run without recompilation. This coexistence of 32-bit legacy applications and 64-bit applications is implemented with a compatibility thunking layer.

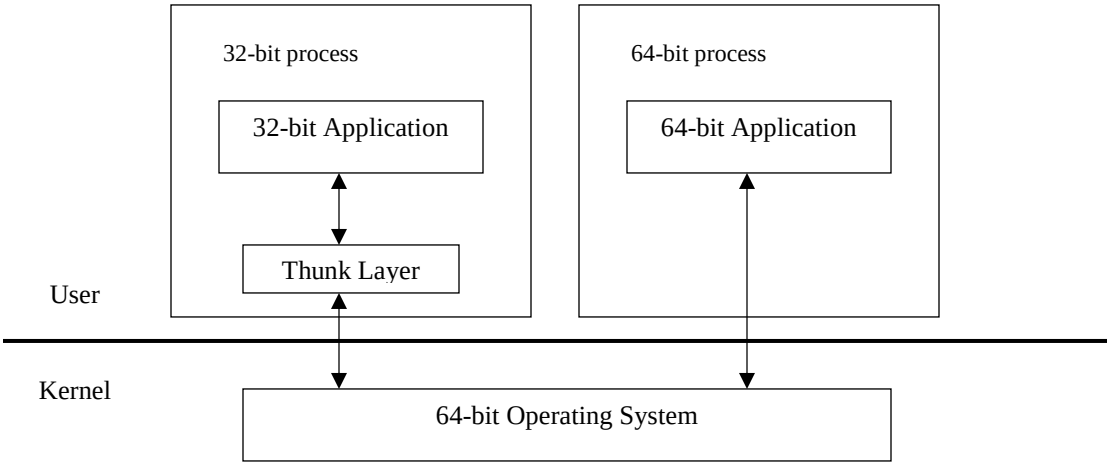

Figure 3-8. AMD x86-64 architecture in compatibility mode

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. The thunk layer implements 32-bit system calls. The thunk layer translates system call parameters, calls 64-bit kernel and translates results returned by the kernel appropriately and transparently for a 32-bit application.

For detailed information on x86-64 architecture, please refer to the following AMD Opteron technical documentation:

http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_7044,00.html

USB, 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 TSF software and the TSF databases for the SLES system. The descriptions are organized according to the structure of the system and describe a 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 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

eSeries servers 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 xSeries, pSeries, iSeries, zSeries, and eServer 325 systems.

xSeries
xSeries 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, 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.

pSeries and iSeries
pSeries and iSeries servers are powered by PowerPC processors. These processors provide three execution modes identified by the PR bit (bit 49) and the HV bit (bit 3) of the processor's Machine State Register. 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, and an HV bit value of 1 and a PR bit value of 1 indicate a user execution mode.

zSeries
zSeries 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, whereas the value of 1 indicates a "problem state" (or user) execution mode.

eServer 325
eServer 325 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 eSeries 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 privilege. Software that runs with hardware privileges includes:

- The base SLES kernel. This is the large body of software that performs memory management file I/O and process management.
- Separately loaded kernel modules, such as **ext3**. 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. All user processes run at least part of the time with hardware privileges, when a user-mode process makes a system call. The execution of the system call switches the current process 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.

4.2 TSF software structure

This section describes the structure of the SLES software that is included in the TSF. The SLES system is a multi-user operating system with a kernel. The kernel runs 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. The following sections provide more detailed descriptions of the kernel and non-kernel TSF architectural subsystems. Each architectural subsystem, which may include multiple programs and supporting data files, is briefly described with respect to the functions it performs.

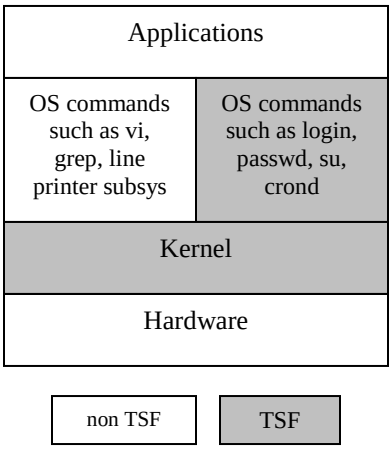

Figure 4-1. TSF & non TSF software

Logical subsystems are concepts described in the Common Criteria. These logical subsystems are the building blocks of the TOE, as described in the Functional Descriptions chapter. 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.

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

- Controlled execution of processes by allowing their creation, maintenance, and termination.
- Scheduling of multiple processes for execution.
- Allocation of private memory for each executing process.
- Configuration of a part of the hard disk as virtual memory.
- Mechanism for storing and retrieving user data from a storage device.
- Access to peripheral devices such as disk drives, printers, and terminals.
- Communication mechanism that allows different processes to communicate with each other.

4.2.1.1 Logical components

The kernel can be thought of as consisting of logical subsystems. These are logical subsystems only; that is, the kernel is a single executable program, but the services it provides allow it to be "broken up" into logical components. These components interact to provide specific functions.

The following schematically describes logical kernel subsystems, their interactions with each other and with the system call interface available from user space.

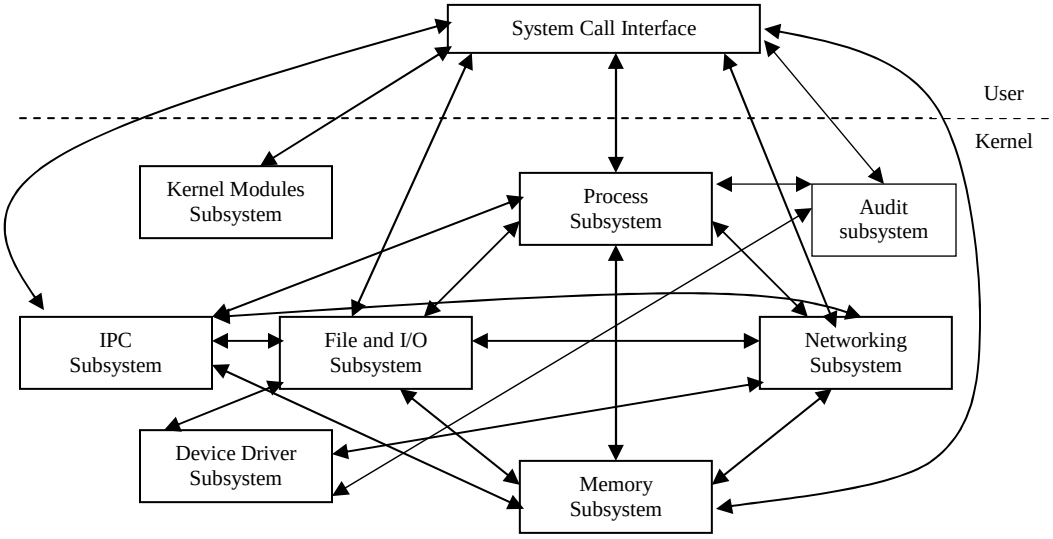

Figure 4-2. Logical kernel subsystems and their interactions

The kernel consists of the following logical subsystems:

File and I/O subystem
Implements all file system object-related functions. Functions include those that allow a process to create, maintain, interact, and delete file system objects, such as regular files, directories, symbolic links, hard links, device-special files, named pipes, and sockets.

Process subsystem
Implements functions related to process and thread management. Functions include those that allow the creation, scheduling, execution, and deletion of process and thread subjects.

Memory subsystem
Implements functions related to the management of a system's memory resources. Functions include those that create and manage virtual memory, including management of page tables and paging algorithms.

Networking subsystem
Implements the UNIX and Internet domain sockets as well as algorithms for scheduling network packets.

IPC subsystem
Implements functions related to inter-process communication mechanisms. 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
Implements an infrastructure to support loadable modules. Functions include those that load and unload kernel modules.

Device driver subsystem
Implements support for various hardware and software devices through a common, device-independent interface.

Audit subsystem
Implements functions related to the recording of security-critical events on the system. Functions include those that hook security relevant system calls to record security critical events, and those that implement the collection and recording of audit data.

4.2.1.2 Execution components

From the perspective of execution, the kernel can be divided into three components: Base Kernel, Kernel Threads, and Kernel Modules.

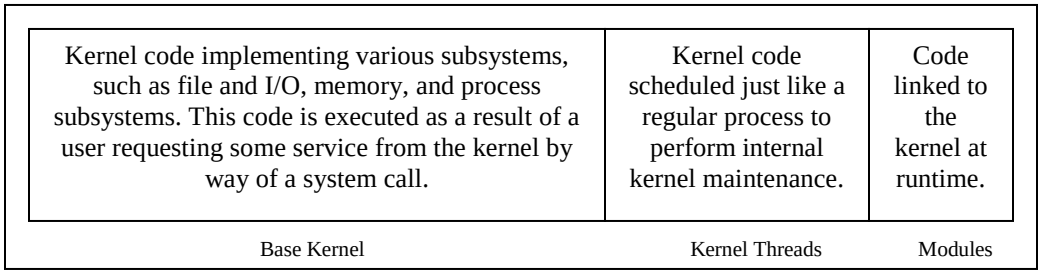

Figure 4-3. Kernel execution components

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

Kernel threads
In order to perform certain routine tasks, such as flushing disk caches and 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 a specific kernel C function. Kernel threads reside in kernel space and only run in the kernel mode. The following lists some of the kernel threads:

- **`keventd`**
  A handler for the process context bottom half. `keventd` executes tasks, created by interrupt handlers, which are queued in the `qt_context` task queue.

- **`kapmd`**
  Handles advanced power management related tasks.

- **`kswapd`**
  Reclaims memory. `kswapd` is activated by the kernel when the number of free page frames for a memory zone fall below a warning threshold.

- **`kupdated`**
  Flushes old, dirty buffers to disk to reduce the risk of file system inconsistencies.

- **`ksoftirq`**
  Executes pending softirqs. Each CPU has its own `ksoftirq` thread. `ksoftirq` checks which `softirqs` are pending, and executes the function to handle them.

- **`kjournald`**
  Manages the logging device journal. `kjournald` periodically commits the current state of the file system to disk and reclaims space in the log by flushing buffers to disk.
- **`bdflush`**

In the 2.4 Linux kernel, `bdflush` kernel thread periodically flushed dirty buffers to disk to reclaim memory. In TOE, which is based on the version 2.6 of the Linux kernel, `bdflush` is deprecated. The system call `bdflush`, which was used to launch the `bdflush` daemon, is now a dummy system call that captures its use by applications by generating an entry in the kernel log.

Kernel threads are created with a call to `kernel_thread()` and can be listed from user space with the command **ps axu**. The kernel threads are shown in square brackets and can be recognized by their virtual memory size (VSZ) of 0; for example [kjournald].

Kernel modules and device drivers
Kernel modules are pieces of object code that can be linked to and unlinked from the kernel at runtime. 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 is used to control an I/O device such as a hard disk, a monitor, or a 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, independent of their underlying implementations.

4.2.2 Non-kernel TSF software

The software that runs without hardware privilege is organized into processes. A process is an active entity, sometimes referred to as a "program in execution" or an "executing image of a program." The non-kernel TSF software consists of trusted programs that are used to implement security functions. 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. The daemon processes that provide TSF functionality include `agetty`, `sshd`, `vsftpd`, `atd`, `cron`, and `auditd`.
- Programs that are executed by an unprivileged user and need access to certain protected databases to complete their work. These programs include `login`, `passwd`, `gpasswd`, `chage`, `su`, `at`, and `crontab`.
- Shared libraries that are used by trusted programs. These libraries include PAM (Pluggable Authentication Module) modules.

4.2.2.1 init

`init` is the parent of all user processes. `init` is handcrafted by the kernel and is the first process that runs in the user space. `init` consults the */etc/inittab* file and spawns processes based on the default run level specified in the */etc/inittab* file.

4.2.2.2 agetty and mingetty

`agetty`, the alternative linux getty, is invoked from */sbin/init* when the system becomes available in a multiuser mode. `agetty` opens a tty port, prompts for a login name, and invokes */bin/login* to authenticate. `mingetty`, the minimal getty for consoles, provides the same functionality as `agetty`. However, unlike `agetty`, which is used for serial lines, `mingetty` is used for virtual consoles.

4.2.2.3 ssh and sshd

`ssh` and `sshd` are client-server pair that allow authorized users to log in from remote systems using secure encrypted communications.

4.2.2.4 vsftpd

`vsftpd` is a server that allows authorized users to transfer files to and from remote systems.

4.2.2.5 crontab and cron

`crontab` and `cron` are a client-server pair that allow the execution of commands on a recurring basis at a specified time. `crontab` creates a file that sets up the time and frequency of execution as well as the command/script to execute. `cron` is the daemon that reads the `crontab` files for all users and performs tasks specified in the `crontab` files on behalf of the user. `cron` is started by the `init` program during system initialization.

4.2.2.6 login

`login` is used when a user signs 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*. `login` prompts for the password and turns off the terminal echo in order to prevent the password from being displayed as it is being typed by the user. `login` 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 up in the */etc/login.defs* file, are checked. 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 a shell specified in the */etc/passwd* file. Please refer to the `login` man page for more detailed information.

4.2.2.7 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 utilizes 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 displaying the password as it is being typed by the user. Please refer to the `passwd` man page for more detailed information.

4.2.2.8 gpasswd

`gpasswd` administers the */etc/group* and */etc/gshadow* files. `gpasswd` allows system administrators to designate group administrators for a particular group. Please refer to the `gpasswd` man page for more detailed information.

4.2.2.9 chage

`chage` allows the system administrator to alter a user's password expiration data. Please refer to the `chage` man page for more detailed information.

4.2.2.10 su

`su` allows a user to switch identity. `su` changes the effective user and group IDs to those of the new user. Please refer to the `su` man page for more detailed information.

4.2.2.11 useradd, usermod and userdel

`useradd`, `usermod`, and `userdel` allow an administrator to add, modify, or delete a user account. Please refer to their respective man pages for more detailed information.

4.2.2.12 groupadd, groupmod and groupdel

`groupadd`, `groupmod`, and `groupdel` allow an administrator to add, modify, or delete a group. Please refer to their respective man pages for more detailed information.

4.2.2.13 at and atd

`at` and `atd` are a client-server pair that allow users to create tasks that are executed at a later time. Unlike `crontab`, `at` reads commands from the standard input and executes them using the user's shell. `atd` 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.

4.2.2.14 atrm

`atrm` removes jobs already queued for execution. `atrm` deletes jobs, whose job numbers are passed to the command line as arguments.

4.2.2.15 ping

`ping` sends the ICMP protocol's mandatory `ECHO_REQUEST` datagram to elicit an `ICMP_ECHO_RESPONSE` from a host or a gateway.

4.2.2.16 chsh

`chsh` allows a user to change his or her login shell. If a shell is not given on the command line, `chsh` prompts for one.

4.2.2.17 chfn

`chfn` allows a user to change his or her finger information. The information, stored in */etc/passwd* file, is displayed by the `finger` command.

4.2.2.18 xinetd

`xinetd` is the super-server that starts other servers that provide network services, such as file transfer, between systems.

4.2.2.19 auditd

The audit daemon reads audit records from the kernel buffer through the audit device and writes them to disk in the form of audit logs.

4.2.2.20 aucat

`aucat` reads the binary audit log files and outputs the records in human-readable format.

4.2.2.21 augrep

`augrep` performs similar function as `aucat`, but it allows an administrative user to optionally filter the records based on user, audit id, outcome, system call or file name.

4.2.2.22 aurun

`aurun` is a wrapper application that allows the attachment of trusted processes to the audit subsystem.

4.2.2.23 audbin

`audbin` is a trusted application that can be used to manage audit log files.

4.2.2.24 stunnel

`stunnel` is a command line tool designed to work as an SSL encryption wrapper between remote clients and local (xinetd-startable) or remote servers.

4.2.2.25 openssl

`openssl` is a command line tool used to setup various cryptography functions used by the Secure Socket Layer (SSL v3) and Transport Layer Security (TSL v1).

4.2.2.26 amtu

`amtu` 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.

4.2.2.27 date

`date` command can be used to print or set the system date and time. Only an administrative user is allowed to set the system date and time.

4.3 TSF databases

The following table identifies the primary TSF databases that are used in the SLES and explains their purpose. The databases are listed as individual files (by pathname or a collection of files). Some are readable by all users; only the root user has write access to all of the TSF databases. Access control is performed by the file system component of the SLES kernel. For more information on the format of these TSF databases, please refer to their respective section 5 man pages.

Database	Purpose
/var/log/lastlog	Stores the time and date of the last successful login for each user.
/var/log/faillog	Stores the time and date of the last failed login attempt for each user.
/etc/inittab	Describes the process started by the init program at different run levels.
/etc/init.d/*	System startup scripts.
/etc/passwd	Stores the login name, UID, primary GID, user name, home directory, and shell for all system users.
/etc/security/opasswd	Stores previously used passwords. It is used by the pam_pwcheck module to prevent users from changing their passwords to previously used passwords.
/etc/group	Stores the group names, supplemental GIDs, and group members for all system groups.
/etc/hosts	Contains hostnames and their address for hosts in the network. The */etc/hosts* file resolves a hostname into an Internet address in the absence of a domain name server. The resolving mechanism works bi-directionally.
/etc/shadow	Defines user passwords in one-way encrypted form, plus additional password characteristics.
/etc/login.defs	Defines various configuration options for the login process.
/etc/securetty	Lists terminals from which root can log in.
/etc/ld.so.conf	Configuration file dynamic linker/loader.
/etc/modprobe.d	Contains 2.6 format component configurations to be included in

Database	Purpose
	modprobe.conf
/etc/modprobe.conf	Configuration file for modprobe. modprobe automatically loads or unloads a module while taking into account its dependencies.
/etc/modprobe.conf.local	Local modprobe component configuration file.
/etc/pam.d/*	Contains the configuration files for PAM. */etc/pam.d* contains one file for each application that performs identification and authentication. Each configuration file contains PAM modules to be used for that application.
/etc/security/pam_pwcheck.conf	Configuration file for the PAM module for password strength checking.
/usr/lib/cracklib_dict.*	Dictionaries for the PAM module for password strength checking.
/etc/security/pam_unix2.conf	Configuration file for the PAM module for traditional password authentication.
/var/spool/cron/tabs/root	Crontab file for the root user.
/var/spool/cron/allow	Lists users that are allowed to submit `cron` jobs. If this file exists, only users listed in the file are allowed to submit `cron` jobs.
/var/spool/cron/deny	Lists users that are not allowed to submit `cron` jobs.
/etc/crontab	`Crontab` file for the system.
/etc/cron.d/*	Cron jobs for the system.
/etc/cron.{weekly hourly daily monthly}/*	Cron jobs to be executed weekly, hourly, daily, and monthly.
/var/spool/atjobs/*	atjobs submitted by users.
/etc/at.allow	Lists users that are allowed to submit `at` jobs. If this file exists, only users listed in the file are allowed to submit `at` jobs.
/etc/at.deny	Lists users that are not allowed to submit `at` jobs.
/etc/ssh/sshd_config	Configuration file for the sshd server.
/etc/sysconfig/*	Contains files that configure various system components such as keyboard and network.
/etc/ftpusers	Contains a list of users who cannot log in via the FTP daemon.
/etc/vsftpd.conf	Configuration file for the Very Secure FTP daemon.
/etc/xinetd.conf	Configuration file for the Extended Internet Services daemon.
/etc/audit/audit.conf	Configuration file for the audit subsystem.
/etc/audit/filter.conf	Configuration file for filter audit records based on input parameters.
/etc/audit/filesets.conf	Configuration file containing list of pathnames to be used to filter audit records.
/etc/stunnel/*.conf	Configuration file for stunnel command
/etc/stunnel/stunnel.pem	File with certificate and private key for stunnel command

Table 4-1. TSF Databases

4.4 Definition of subsystems for the CC evaluation

Previous sections define various logical subsystems that make up the SLES system. One or more of these logical subsystems 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.

Kernel subsystems

The following sections describe the subsystems implemented as part of the SLES kernel.

4.4.1 File and I/O

This subsystem includes the file and I/O management kernel subsystem only.

4.4.2 Process control

This subsystem includes the process control and management kernel subsystem.

4.4.3 Inter-process communication

This subsystem includes the inter-process communication kernel subsystem.

4.4.4 Networking

This subsystem contains the kernel networking subsystem.

4.4.5 Memory management

This subsystem contains the kernel memory management subsystem.

4.4.6 Kernel modules

This subsystem contains routines in the kernel that create an infrastructure to support loadable modules.

4.4.7 Device drivers

This subsystem contains the kernel device driver subsystem.

4.4.8 Audit

This subsystem contains the kernel auditing subsystem.

Trusted process subsystems

The following section describes the subsystems implemented as trusted processes.

4.4.9 System initialization

This subsystem consists of the boot loader (`grub`) and the `init` program.

4.4.10 Identification and authentication

This subsystem contains the `su`, `passwd`, and `login` trusted commands as well as the `agetty` trusted process. This subsystem also includes Pluggable Authentication Module (PAM) shared library modules.

4.4.11 Network applications

This subsystem contains `vsftpd` and `sshd` trusted processes, which interact with PAM modules to perform authentication. It also includes the `xinet` daemon (`xinetd`) and the `ping` program.

4.4.12 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

4.4.13 Batch processing

This subsystem contains the trusted programs used for the processing of batch jobs. They are:

- `at`, `atd, atrm`
- `crontab`, `cron`

4.4.14 User level audit subsystem

This subsystem contains the portion of the audit system that lies outside the kernel. This subsystem contains `auditd` trusted process, which reads audit records from kernel buffer and transfer them to on-disk audit logs, trusted audit management commands `aucat`, `augrep`, `aurun` and `audbin`, audit logs, audit configuration files, and audit libraries.

5 Functional descriptions

The SLES kernel structure, its trusted software, and its TSF databases provide the foundation for the descriptions in the Functional Architectural Subsystems Description section of this document.

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 IPC subsystem, the process subsystem, the audit subsystem, and the device drivers.

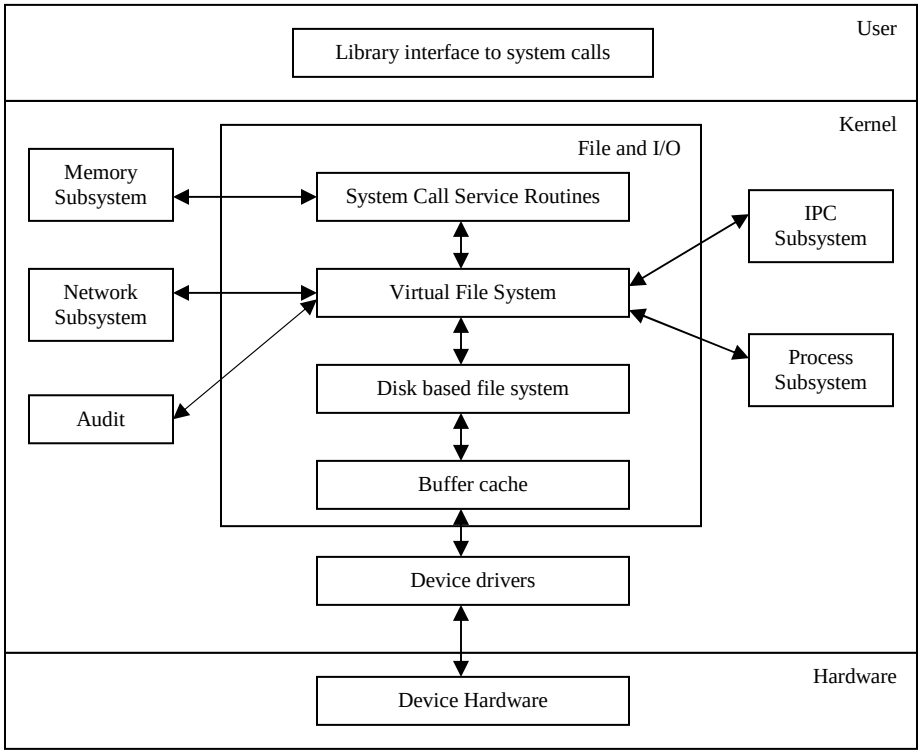

Figure 5-1. File and I/O subsystem and its interaction with other subsystems

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

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 Control policy. This section also highlights how file system data and metadata are allocated and initialized to satisfy the object reuse requirement.

For more detailed information on other aspects of File and I/O management, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

http://acl.bestbits.at/about-acl.html
Posix 1003.1e document at http://wt.xpilot.org/publications/posix.1e

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 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

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 follows.

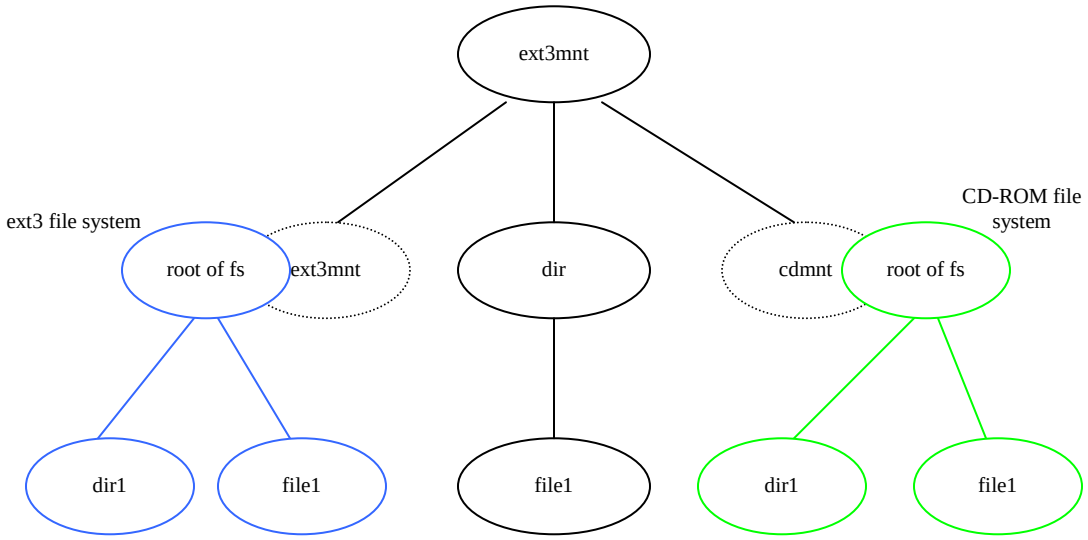

Figure 5-2. ext3 and CD-ROM file systems

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

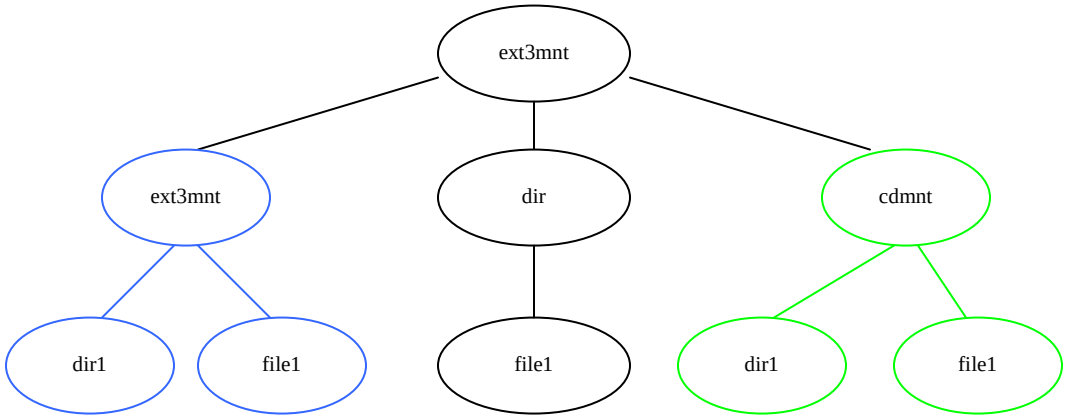

Figure 5-3. ext3 and CD-ROM file systems

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. For example, see the following diagram.

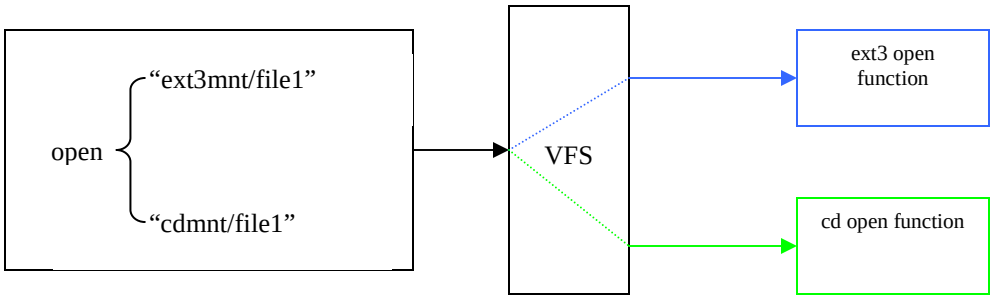

Figure 5-4. Virtual File System

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 security relevant 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 a mounted file system, such as file system type, block size, maximum size of files, and `dentry` (described below) object of the mount point.

`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.

`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. `file` exists only in kernel memory during the period when each process accesses a 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, and 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 security relevant operations of pathname translation and file system mounting.

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 file creation or file deletion), checks to make sure that the process has write access to the directory being modified. Directories cannot be written directly.

Access checking in VFS is performed while an inode is derived from the corresponding pathname. 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 process's current directory, 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.

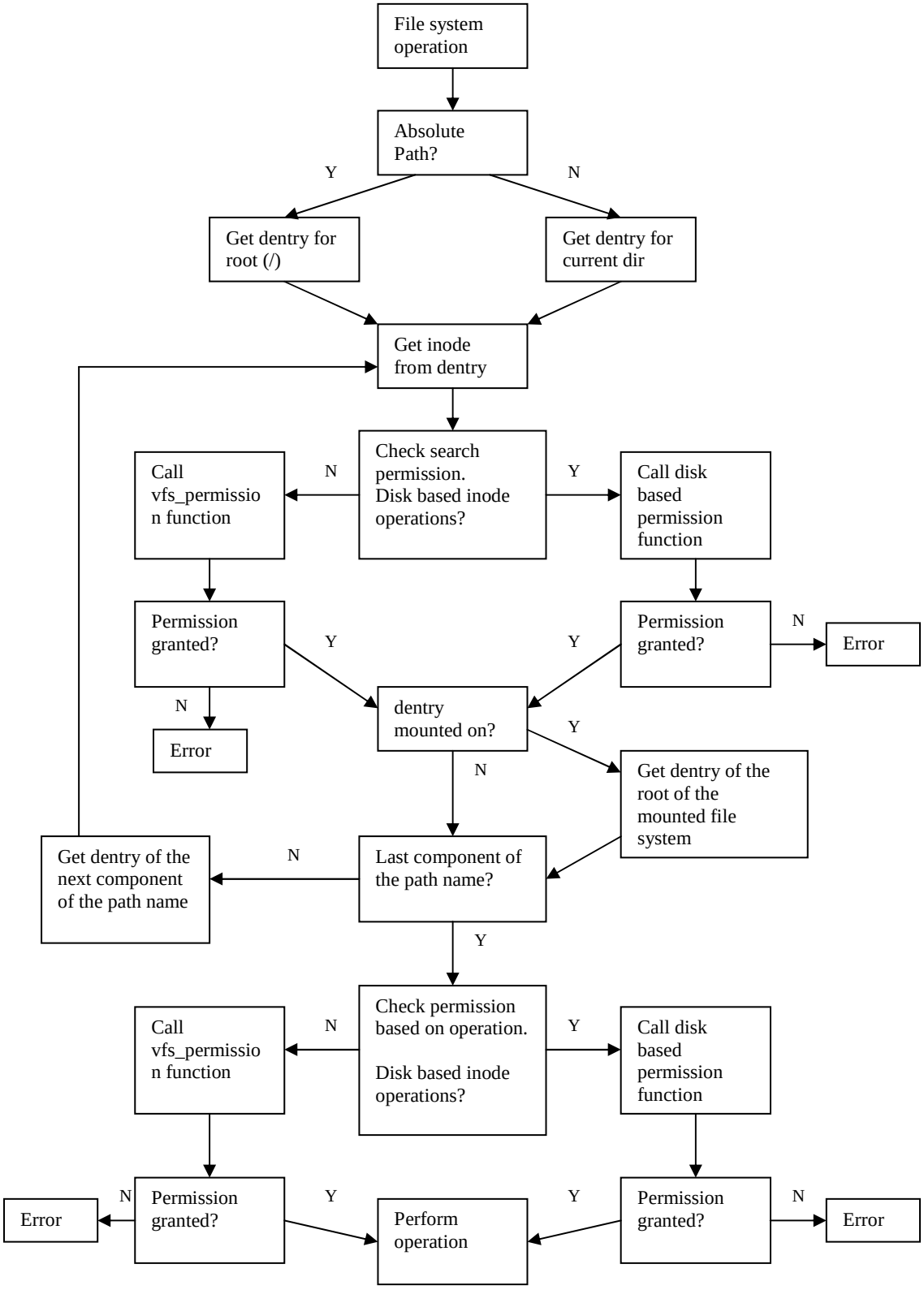

Figure 5-5. VFS pathname translation and access control checks

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, ".", ".." and extra "/" characters in the pathname. Even though these objects complicate the logic of the lookup routine, the access check mechanism remains the same.

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 `inode`'s operation vector (which contains pointers to valid inode operation routines) is set, each call to `permission()` is diverted to the disk-based file system-specific permission call; otherwise, `vfs_permission()` is called to ensure that the process has the appropriate permission.
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 `inode`'s operation vector is set, the call to `permission()`is diverted to the disk-based file system-specific permission call; otherwise, `vfs_permission()` is called to ensure that the process has the appropriate permission.
4. If the user is authorized to create a file in this directory, `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 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 `dentry` object to `inode` object. The following diagram shows the simplified linkage.

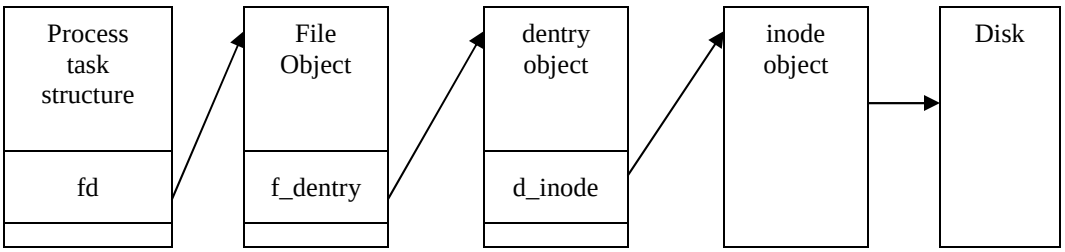

Figure 5-6. VFS data structures and their relationships with each other

Another example of a file system operation is a `write()` system call to write to a file that was opened for writing. The following list shows the call sequence of a `write`() call.

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

The `write`() system call in VFS is very straightforward because access checks are already performed by `open()`.

5.1.1.2 File system mounting

File systems are mounted by an administrator using the `mount()` system call. The `mount()` system call provides the kernel with the file system type, the pathname of the mount point, the pathname of the block device that contains the file system, flags that control the behavior of the mounted file system, and 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 data structure of type `vfsmount`. 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. For detailed information on the mount process, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

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. The following lists the security-relevant mount flags and their implications on access control.

`MS_RDONLY`
The file system is mounted in read only mode. Write operations are prohibited for all files irrespective of their mode bits. Only device special files may be writable.

`MS_NOSUID`
`suid` and `sgid` bits on executables are ignored by the kernel 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.

5.1.2 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 study 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 is a robust and efficient file system that supports automatic consistency checks, immutable files, preallocation of disk blocks to regular files, fast symbolic links, Access Control Lists, and 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 (EA). ACLs provide a flexible method for granting or denying access to a directory or a file that is granular down to an individual user. Extended attributes provide a mechanism for setting special flags on a directory or a file. Some of these improve the system's usability while others improve its security. The following diagram illustrates the format of an ext3 inode.

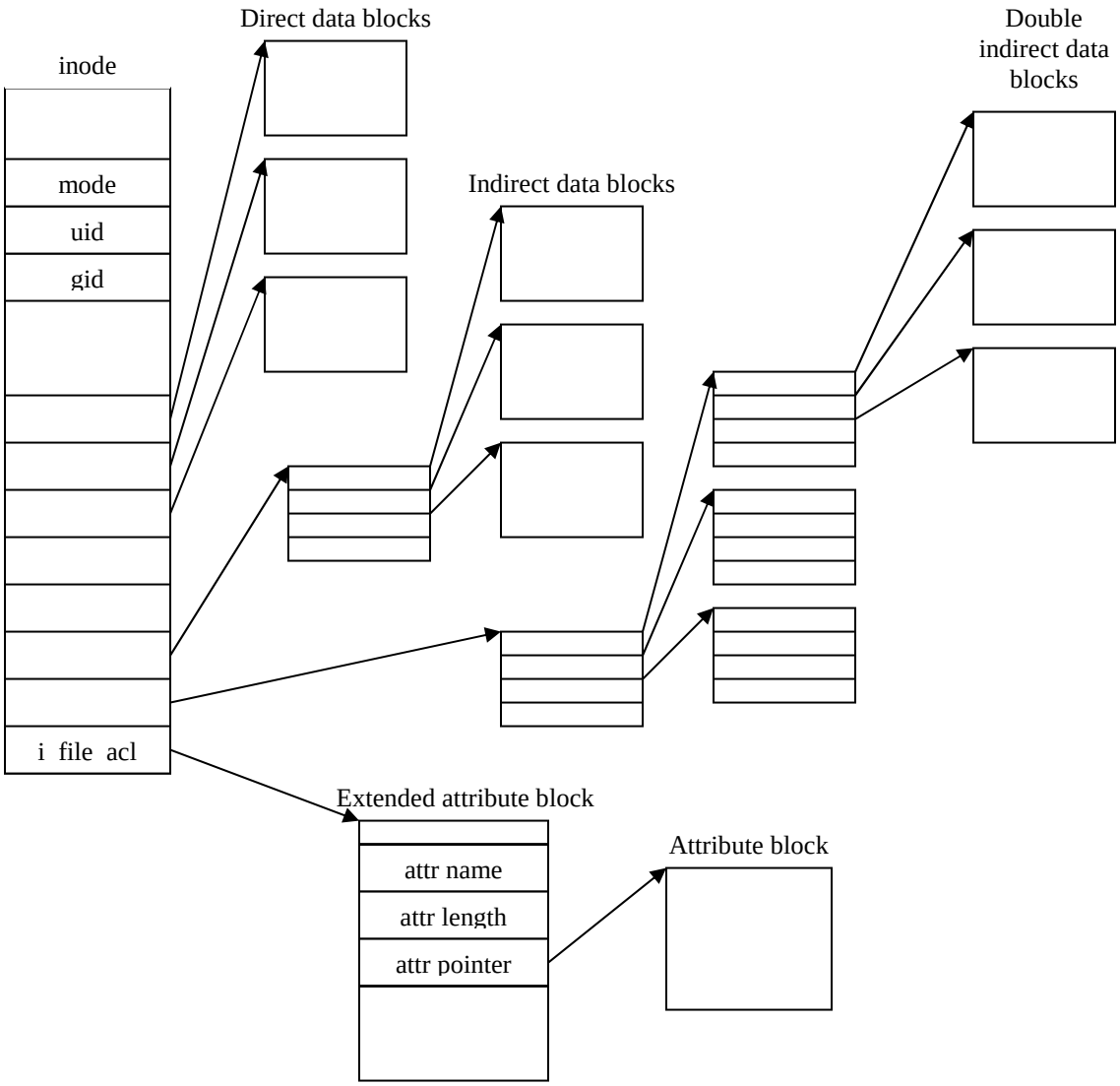

Figure 5-7. Security attributes, extended security attributes and data blocks for ext3 inode.

Access Control Lists provide a way of extending directory and file access restrictions beyond the traditional owner, group, and world permission settings. For more details on the ACL format, please refer to section 5.1.3 of this document.

Extended attributes are stored on disk blocks allocated outside of an inode. Security-relevant extended attributes 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 be modified in append mode. The "`append only`" attribute is useful for system logs.

The following data structures and `inode` operations illustrate how Discretionary Access Control and object reuse are performed by the ext3 file system.

`ext3_super_block`
The on-disk counterpart of the `superblock` structure of VFS, `ext3_super_block` stores file system-specific information such as 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` bit map and the block number of the block bitmap

`ext3_inode`
The on-disk counter part 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`
The structure that describes an extended attribute entry. `ext3_xattr_entry` stores information such as attribute name, attribute size, and the disk block that stores the attribute. Access Control Lists 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's `real_lookup()` calls the disk-based file system's lookup routine through the `inode` operation vector. `ext3_lookup()` calls `ext3_find_entry()` to locate an entry in a specified directory with the given name.

`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. `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.

The following illustrates how new data blocks are allocated and initialized for an ext3 file.

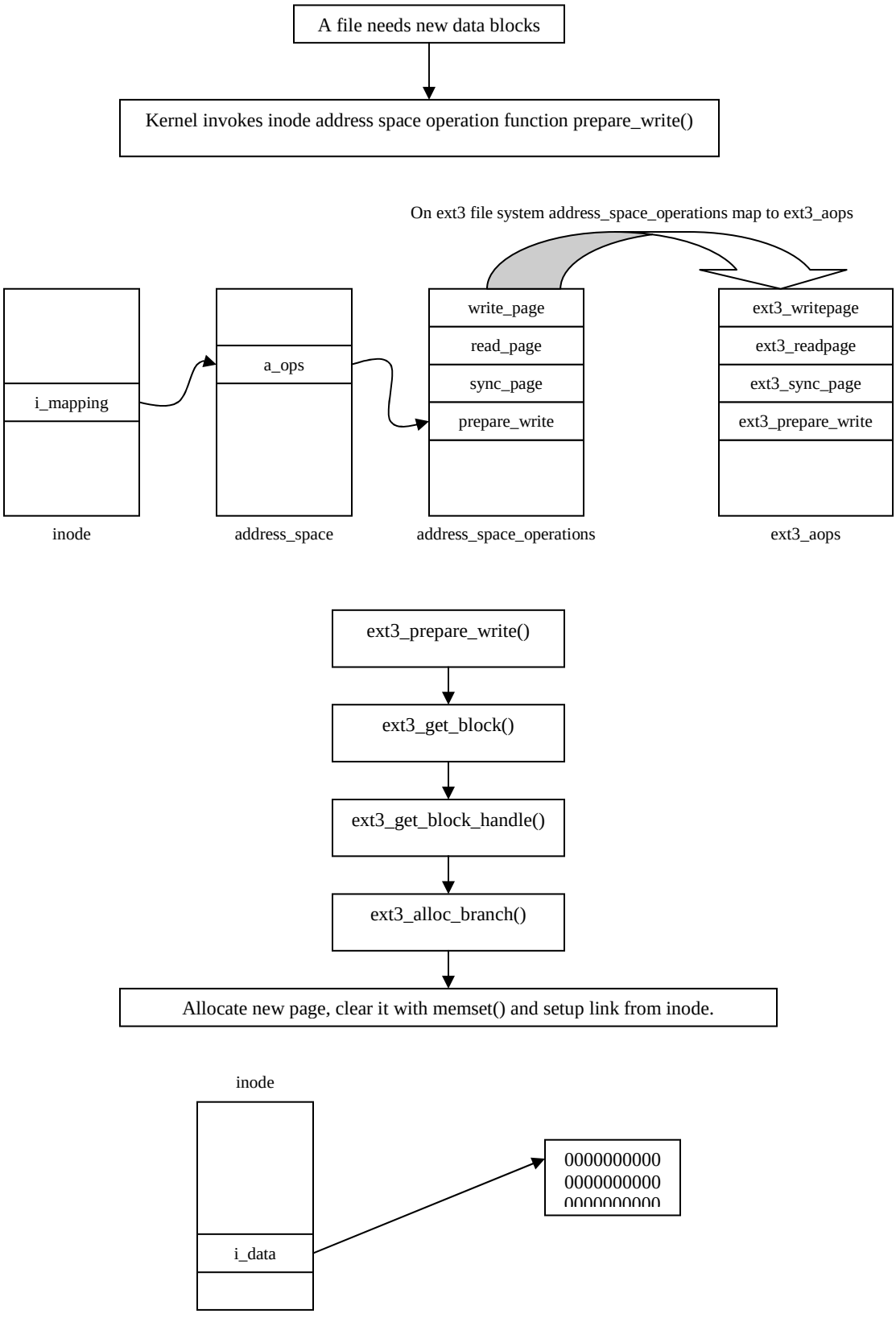

Figure 5-8. Data block allocation and object reuse handling on ext3 file system

ext3_permission()
This is the entry point for all Discretionary Access Check. This routine is invoked when VFS calls to `permission()` routine are diverted based on the ext3 inode operation vector `i_op`.

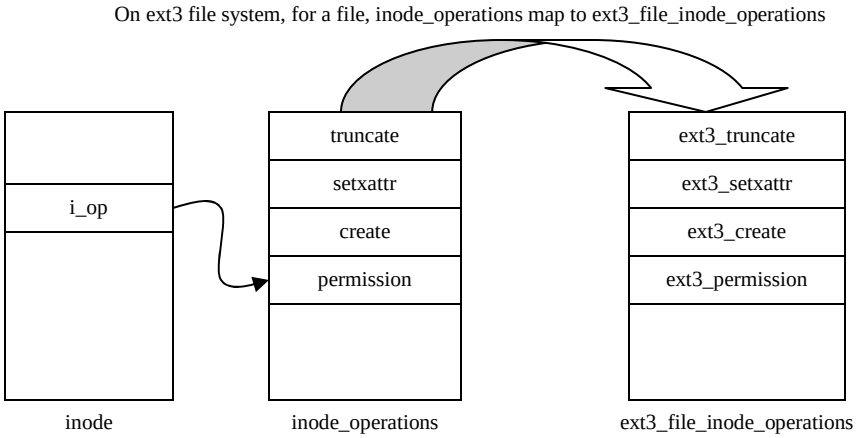

Figure 5-9. Access control on ext3 file system

Similarly, for directory, symlink and special-file object type, inode operations map to `ext3_dir_inode_operations`, `ext3_symlink_inode_operations`, and `ext3_special_inode_operations`, respectively.

ext3_truncate()
This 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 operation vector `i_op`. This routine prevents the truncation of inodes whose extended attributes mark them as "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. Please refer to the following HOWTO document on The Linux Documentation Project for a detailed specification of the ISO 9660 file system:

http://usr/share/doc/howto/en/html_single/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. 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 the process *`fsuid`* field.

isofs_sb_info
The CD-ROM file system super block `isofs_sb_info` stores file system-specific information, such as 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 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()`

`isofs_lookup()` is called when the pathname translation routine is diverted from the VFS layer to the isofs layer. `isofs_lookup()` sets up the `inode` operation vector from the superblock `s_root` field and then invokes `isofs_find_entry()` to retrieve the object from the CD-ROM.

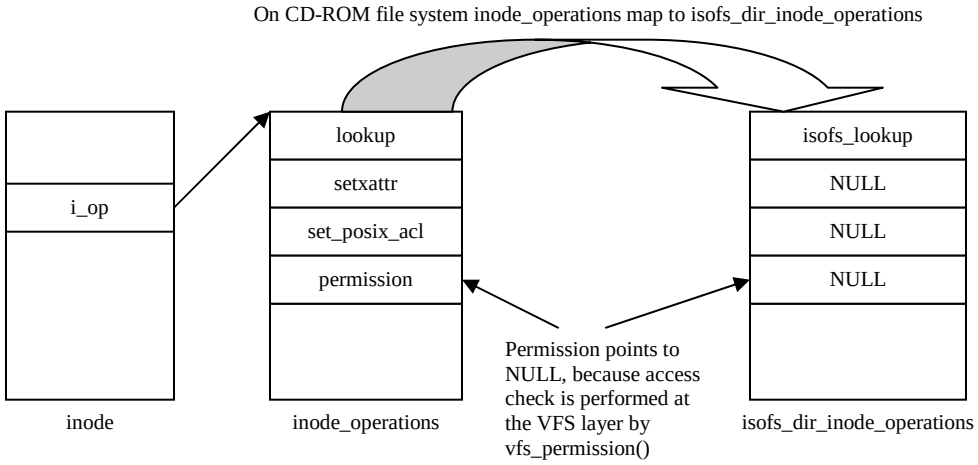

Figure 5-10. File lookup on CD-ROM file system.

5.1.3 proc file system

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 belonging to other processes. Permissions for files in */proc* cannot be changed; they are determined by the kernel. 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.

5.1.4 sysfs file system

The sysfs file system is a special file system that allows system programs and administrators to manipulate the non-process related data structures of the kernel. Process related data structures are accessed through the proc file system, while non-process related data structures, mainly device driver characterstics, are accessed through sysfs file system. The sysfs file system is mounted at */sys* using the file system type of "sysfs", and provides Virtual File System access to kernel object data structures. An administrator can change kernel object parameters, by editing files in */sys*. For each active kernel object, the kernel creates a directory in the */sys* directory. 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. Permissions for files in */sys* cannot be changed; they are determined by the kernel. Because files in */sys* are not real disk-based files with user data, there is no object reuse issue.

5.1.5 tmpfs file system

The tmpfs file system is a special virtual file system that provides file system interface to memory regions. The tmpfs is not persistant across boots and provides fast, temporary, ram-based file system. On a typical system tmpfs is mounted on */dev/shm*; however it can be mounted on other directories just like any other file system. Unlike ext3 or isofs, tmpfs is not a disk-based file system. tmpfs uses vfs pathname translation function `vfs_permission` and inode attribute change function `inode_change_ok` to perform discretionary access control checks. Object reuse is handled by `shmem_getpage` function, which is called anytime new memory pages are allocated and associated with files within this shared memory virtual file system. `shmem_getpage` calls `shmem_alloc_page` to allocate and zero out, using explicit call to `memset`, new memory pages.

5.1.6 devpts file system

The *devpts* file system is a special file system that provides pseudo terminal support. Pseudo terminals are implemented as character devices. A pseudo terminal is represented by a pair of character device-special files, one corresponding to the master device and the other to the slave device. The slave device provides a terminal interface. Instead of a hardware interface and associated hardware supporting the terminal functions, the interface is implemented by a process that manipulates the master device of the pseudo terminal. 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 master device */dev/ptmx*. The system then makes available to the process <number>, as 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.

5.1.7 Discretionary access control

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 option ***grpid,*** the object takes the group ID of the directory in which it is created; otherwise (the 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 `vfs_permission()`, which performs access checks for the ISO9660, procfs, and devpts file systems, and `ext3_permission()`, which performs access checks for the ext3 disk-based file system. 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.

5.1.7.1 vfs_permission()

`vfs_permission()` implements standard UNIX permission bits to provide DAC for file system objects for the procfs, the devpts, and the ISO9660 file systems. 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 CD-ROM) is always rejected. Each subject's access to an object is defined by some combination of these bits:

- `rwx` symbolizing read/write/execute
- `r-x` symbolizing read/execute
- `r--` symbolizing read
- `---` symbolizing 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 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 = 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 process's effective UID is not zero, then the access attempt is denied.

5.1.7.2 ext3_permission()

`ext3_permission()` enforces POSIX Access Control Lists (ACLs). ACLs are created, maintained, and used by the kernel. For more detailed information on the POSIX ACLs, please refer to the following:

http://acl.bestbits.at
http://wt.xpilot.org/publications/posix.1e

An ACL entry contains the following information:

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

ACL Tag Types
The following tag types exist:

1. ACL_GROUP
   An ACL entry of this type defines access rights for processes whose file system group ID or any supplementary group IDs match the one in the ACL entry qualifier.
2. ACL_GROUP_OBJ
   An ACL entry of this type 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.
3. ACL_MASK
   An ACL entry of this type defines the maximum discretionary access rights for a process in the file group class.

4. ACL_OTHER
   An ACL entry of this type defines access rights for processes whose attributes do not match any other entry in the ACL.
5. ACL_USER
   An ACL entry of this type defines access rights for processes whose file system user ID matches the ACL entry qualifier.
6. 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.

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

ACL permissions
The permission that can be defined in an ACL entry is: read, write, and execute/search.

Relation with file permission bits
An ACL contains exactly one entry for each of the ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER tag type (called the "required ACL entries"). An ACL may have between zero and a defined maximum number of entries of the type ACL_GROUP and ACL_USER. An ACL that has only the three required ACL entries is called a "minimum ACL." ACLs with one or more ACL entries of type ACL_GROUP or ACL_USER are called an "extended ACL." The standard UNIX file permission bits, as described in the previous section, are represented by the entries in the minimum ACL. The owner permission bits are represented by the entry of type ACL_USER_OBJ. The entry of type ACL_GROUP_OBJ represents the permission bits of the file group. The entry of type ACL_OTHER 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.

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

Default ACLs and ACL inheritance
A default ACL is an additional ACL, which may 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.

ACL representations and interfaces
ACLs are represented in the SLES kernel as extended attributes. The SLES 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 calling system calls directly, 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 SLES 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.

ACL enforcement
ACLs are used by the `ext3_permission()` function to enforce Discretionary Access Control. `ext3_permission()` calls `__ext3_permission()`, which goes through the following steps:

1. Performs sanity checks such as "no write access if read-only file system" and "no write access if the file is immutable."
2. Calls `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.
3. Invokes `posix_acl_permission()`, which 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.

4. `posix_acl_permission()` 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 attempted execute access only if an execute bit is set for owner, group, or other.

For example, consider a file */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 user root and readable by users belonging to group root. 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 **setfacl** command allows a user "john" read access to this file even if "john" doesn't belong to group root.

```
#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 group root, he is allowed read access to the file */aclfile*.

5.2 Process control and management

A process is defined as 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. It interacts with the memory subsystem, the network subsystem, the file and I/O subsystem, audit subsystem, and the IPC subsystem.

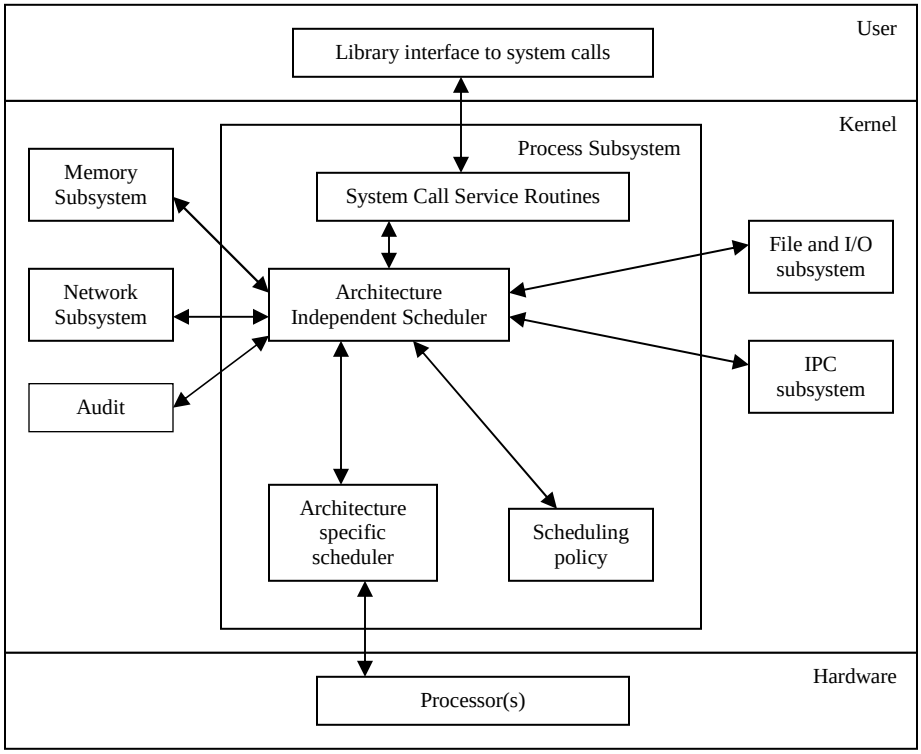

Figure 5-11. Process subsystem and its interaction with other subsystems

The kernel views a process as a subject. A subject is an active entity that can access and manipulate data and data repositories called objects, to which system resources, such as CPU time and memory are allocated. A process is managed by the kernel through a number of data structures. These data structures are created, manipulated, and destroyed to give processes "life."

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. For more

detailed information, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

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/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 referred to as “processes” for better readability.

For each process, the kernel maintains a process descriptor with the `task_struct` structure. The structure’s fields include the process priority, whether the process is running on a CPU or blocked on an event, what address space has been assigned to the process, which files the process is allowed to access, and security relevant credentials fields such as:

- `uid` and `gid`, which describe the process’s user ID and group ID.
- `euid` and `egid`, which describe the process’s effective user ID and effective group ID.
- `fsuid`, `fsgid`, which describe the process’s file system user ID and file system group ID.
- `suid`, `sgid`, which describe the process’s saved user ID and saved group ID.
- `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.

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

Please refer to */usr/src/include/linux/sched.h* for information on other `task_struct` fields.

The following figure schematically shows the `task_struct` structure with fields relevant for access control.

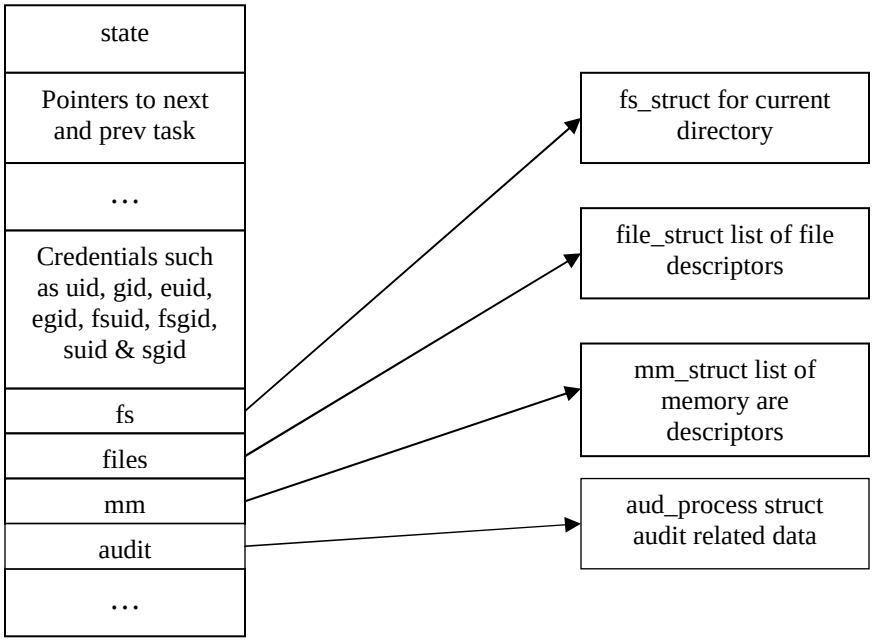

Figure 5-12. The task structure

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/destruction

The SLES kernel provides two system calls for creating a new process: `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," described in section 5.2.3, the object reuse requirement is satisfied. `vfork()` differs from `fork()` by sharing the address space of its parent. To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program. The child process inherits the parent's security-relevant credentials, such as `uid`, `euid`, `gid`, and `egid`. Because these credentials are used for access control decisions in the DAC policy, the child is given the same level of access to objects as the parent. The child's credentials change when it starts executing a new program or issues suitable system calls, which are listed as follows:

- `setuid()` and `setgid()`
  Sets the effective user/group ID and the file system user/group ID of the current process. If the effective user ID of the caller is root, the real and saved user/group IDs are also set.

- `seteuid()` and `setegid()`
  Sets the effective user/group ID and the file system user/group ID of the current process. Normal user processes may only set the effective user/group ID and file system user/group ID to the real user/group ID, the effective user/group ID, or the saved user/group ID.

- `setreuid()` and `setregid()`

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

- `setresuid()` and `setresgid()`

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

- `setfsuid()` and `setfsgid()`

  Sets the user/group ID that the SLES kernel uses to check for all accesses to the file system. Normally, the value of `fsuid/fsgid` shadows the value of the effective user/group ID. `fsuid` and `fsgid` are used by non-disk-based file systems such as NFS. `setfsuid/setfsgid` only succeeds if the caller is an administrator, or if `fsuid/fsgid` matches either the real user/group ID, effective user/group ID, saved set-user/group-ID, or the current value of `fsuid/fsgid`.

- `execve()`

  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 process's Page Tables. `execve()` 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 5.5.3. `execve()` can also alter the process's credentials if the executable file's `setuid` bit is set. If the `setuid` bit is set, the current process's `euid` and `fsuid` are set to the identifier of the file's owner.

  For more details on kernel execution of the `execve()` call, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

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 process's task structure; the rest is saved in the process's kernel mode stack, which allows for the separation needed for a clean switch. In a three-step process, the switch is performed by:

- installation of a new address space

- switching the Kernel Mode Stack
- switching the hardware context

For a more detailed description of process context switching, please refer to the *SLES Low Level Design,* by Janak Desai, George Wilson, and Michael Halcrow.

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 section 5.5.2.

5.3 Inter-process communication

The SLES kernel provides a number of inter-process communication 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 Discretionary Access Control and object reuse handling. For more detailed information, please refer to the *SLES Low Level Design,* by Janak Desai, George Wilson, and Michael Halcrow.

5.3.1 Pipes

Pipes allow the transfer of data in a first-in-first-out (FIFO) manner. Unnamed pipes are created with the `pipe()` system call. 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 and the other for writing. It is the process's responsibility to use the appropriate file descriptor for reading and writing.

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. `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, the way `pipe_read()` and `pipe_write()` are

written it is not possible to read beyond what is written by the write channel. Therefore, there are no object reuse issues.

`pipe_read()`
Invoked through the `read()` system call, `pipe_read()` reads the pipe buffer pointed to by the `base` field of the `pipe_info_structure`.

`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 who share file descriptors, there are no Discretionary Access Control issues.

5.3.2 Named pipes (FIFO)

A FIFO 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 a FIFO's internal implementation is identical to that of the unnamed pipe. Both use the same data structure, `pipe_inode_info,` and routines `pipe_read()` and `pipe_write()`. The only differences are that FIFOs are visible on the system directory tree and are a bi-directional communication channel.

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 system call `mknod()`, as follows:

1. `mknod()` uses the path name translation routines to obtain the `dentry` object of the directory where FIFO is to be created and then invokes `vfs_mknod()`.
2. `vfs_mknod()` 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 inode file operation vector 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 allocation and initialization of inode object is done by the disk-based file system inode allocation routine; thus, object reuse is handled by the disk-based file system.

5.3.2.2 FIFO open

A call to VFS system call `open()` performs the same operation as it does for device special files. Regular Discretionary Access Checks 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. `init_special_inode()` sets the inode's file operation vector `i_fop` 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 Discretionary Access Control and 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 IPC mechanism's "control" system call. 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 Discretionary Access Control. The root user is allowed to override Discretionary Access Control setup through access permissions.
- 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`

Describes the security credentials of an IPC resource with the field `p,` which is a pointer to the resource's credential structure.

`kern_ipc_perm`

The credential structure for an IPC resource with fields such as `key`, `uid, gid, cuid, cgid, mode,` and `seq`. `uid` and `cuid` represent the owner and creator user ID. `gid` and `cgid` represent the owner and creator group ID. `mode` represents the permission bit mask and `seq` identifies the slot usage sequence number.

5.3.3.2 Common functions

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

`ipc_alloc`()

`ipc_alloc()` is invoked from the initialization functions of all three IPC resources to allocate storage space for the IPC resources' respective arrays of IPC resource descriptors. The IPC resource desciptors are pointed to by the `ipc_ids` data structure field, `entries`. 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 is used by the kernel for its internal purposes.

`ipcperms()`
: `ipcperms()` is called when a process attempts to access an IPC resource. Access to the IPC resource is granted based on the same logic as that of regular files, using the object's owner, group, and access mode. The only difference is that the IPC resource's owner and creator 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.

`msg_queue`
: Describes the structure of a message queue with fields such as `q_perm` (which points to the `kern_ipc_perm` data structure), `q_stime` (which contains 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).

`msg_msg`
: Describes the structure 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 message list) and next (which points to `msg_msgseg` corresponding to the next page frame containing the message).

`msgget()`
: The function invoked to create a new message queue or to get a descriptor of an existing queue based on a key. The newly created message queue's credentials are initialized from the creating process's credentials.

`msgsnd()`
: `msgsnd` is a function that is invoked to send a message to a message queue. Discretionary Access Control 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 kernel's buffer space using the `kmalloc()` and `GFP_KERNEL` flag. Thus, no special object reuse handling is required.

`msgrcv()`
: `msgrcv` is a function that is invoked to receive a message from a message queue. Discretionary Access Control is performed by invoking the `ipcperms()` function.

`msgctl()`
: `msgclt` is a function that 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. Discretionary Access Control 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.

`sem_array`
: Describes the structure and state information for a semaphore object. `sem_array` contains fields, such 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).

`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()`.

`semop()` and **`semtimedop()`**
: `semop` is a function that is invoked to perform atomic operations on semaphores. `semtimedop` is similar to `semop`, except that it provides an expiration time for cases where the calling process may sleep. Discretionary Access Control is performed by invoking the `ipcperms()` function.

`semctl()`
: `semctl` is a function that is invoked to set attributes of, query status of, or delete a semaphore. A semaphore is not deleted until the process waiting for a semaphore has received it. Discretionary Access Control 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 data structure `shmid_kernel` and functions `shmat()`, `shmdt()`, `shmget()`, and `shmctl()`.

`shmid_kernel`
: Describes the structure and state information of a shared memory region with fields such as, `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).

`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.

`shmat()`
: `shmat` is invoked by a process to attach a shared memory region to its address space. Discretionary Access Control is performed by invoking the `ipcperms()` function. The pages are added to a process with the "Demand Paging" technique described in section 5.5.3. Hence, the pages are dummy pages. The function adds a new memory region to the process's address space, 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, as described in section 5.5.3, satisfying the object reuse requirement.

shmdt()
: shmdt is invoked by a process to detach a shared memory region from its address space. Discretionary Access Control is performed by invoking the ipcperms() function.

shmctl()
: shmctl is a function that is invoked to set attributes of, query status of, or delete a shared memory region. A shared memory segment is not deleted until the last process detaches it. Discretionary Access Control 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 via the kill() system call, or the kernel can deliver the signals internally. Events that cause a signal to be generated include: keyboard interrupts via the interrupt, stop or quit keys, exceptions from invalid instructions, and termination of a process. Signal transmission can be broken up into the following 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 by starting the execution of a designated signal handler.

Signal transmission does not create any user-visible data structures. Therefore, there are no object reuse issues. However, signal transmission does raise access control issues. This subsection describes relevant data structures and algorithms used to implement discretionary access control. For more detailed information on the design and implemenation of signal generation and signal transmission, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

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. Each process's task_struct contains fields that designate the real and effective user ID of the process. 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() 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 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 (internet domain) or on a single machine (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 port number for the 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 (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.

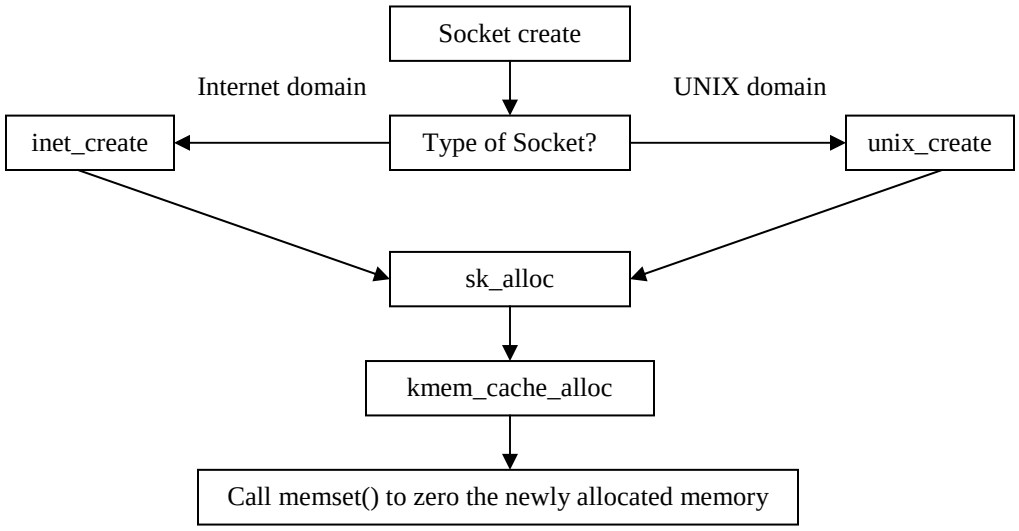

Figure 5-13. Object reuse handling in socket allocation

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 it.

For a more detailed description of client-server communication methods and the access control performed by them, please refer to section 5.10.

5.4 Network subsystem

A network subsystem provides a general-purpose framework within which network services are implemented. It interacts with the file and I/O subsystem, the memory subsystem, the process subsystem, the IPC subsystem, and the device drivers.

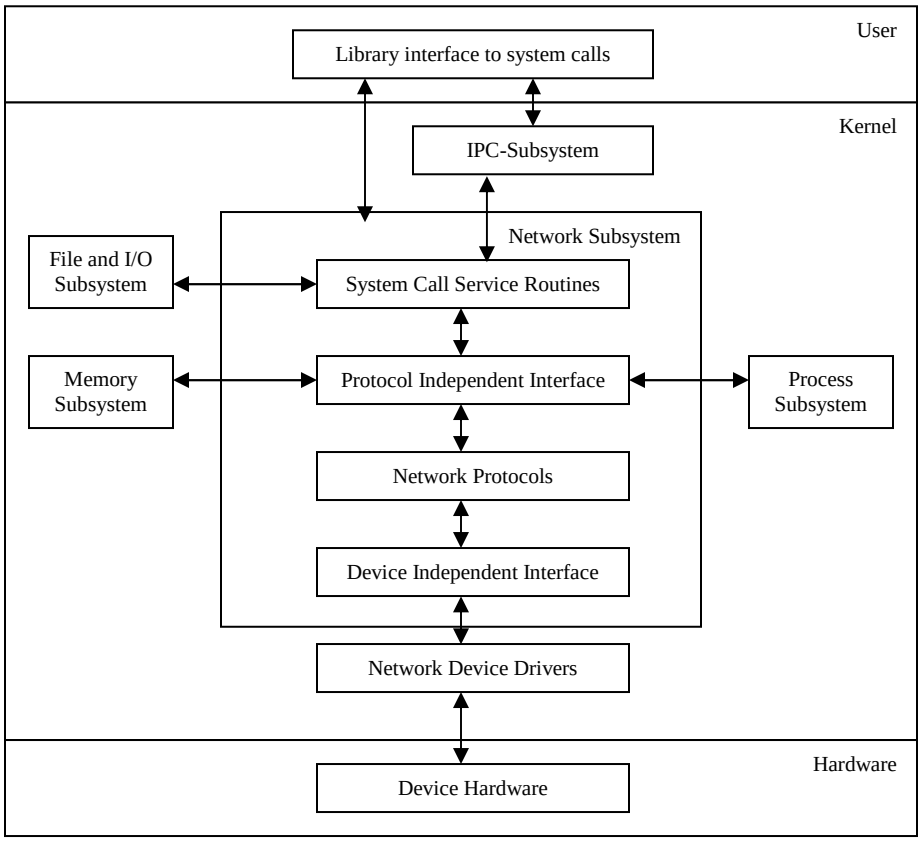

Figure 5-14. Networking subsystem and its interaction with other subsystems

Network services include transmission and reception of data, network independent support for message routing, and network independent support for application software. The network subsystem interfaces to system calls via the IPC subsystem and the socket interface described there. 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 detailed information on the networking subsystem, please refer to the following:

Internetworking with TCP/IP, by Douglas E. Comer & David L. Stevens
SLES Low Level Design, by Janak Desai, George Wilson, and Michael Halcrow
IBM Redbook “TCP/IP Tutorial and Technical Overview”, by Adolfo Rodriguez, et al.
http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf

5.4.1 Overview of network stack

This section describes the path of a network packet within a Linux environment. There are five major layers that comprise the Linux Network Architecture: application, transport, network, link, and physical.

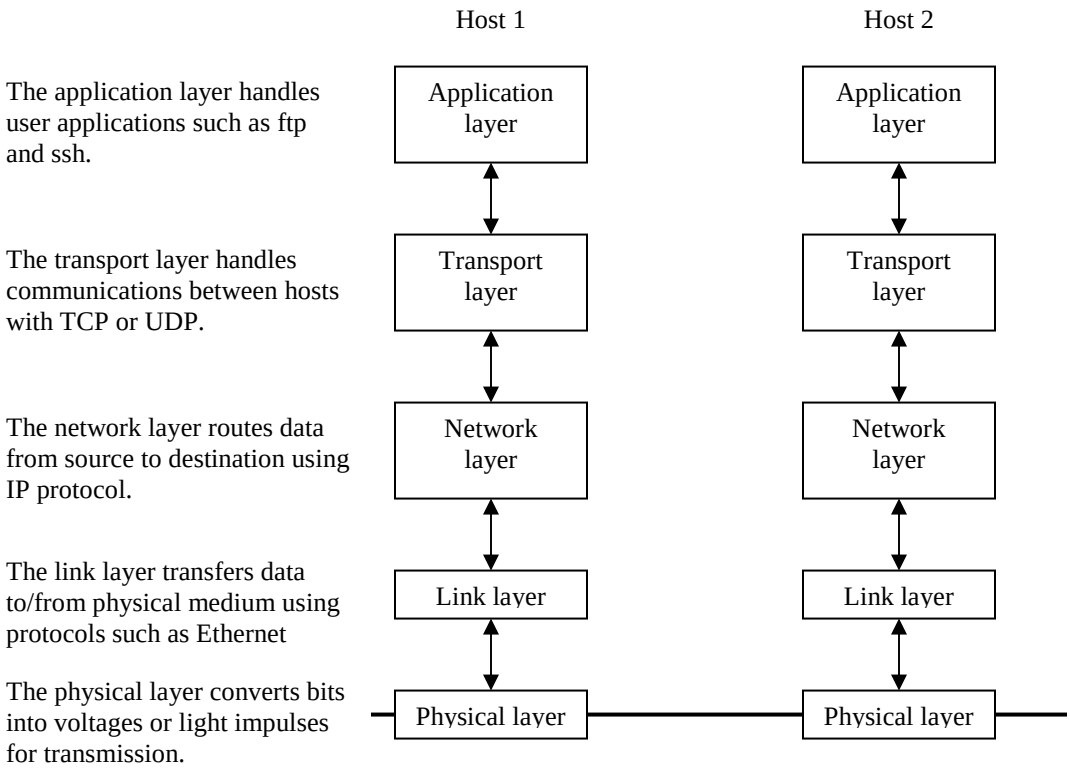

Figure 5-15. Network stack and protocols that operate at each level

The process of an outgoing packet begins at the application level, where such services as `ftp` and `ssh` generate traffic and send a packet of information to the transport layer.

The transport layer is composed of a protocol, which in most cases is either TCP or UDP. These two protocols provide the basic service of addressing packets to various ports in the system, and also provide further services especially in the case of TCP. See the sections below on each particular protocol for further detail.

Once the packet has been routed to the proper port, the network layer takes over. The standard network protocol on Linux is IP. The main role of IP is to check whether a packet is to remain on the host or if it needs forwarding to an outside system. When necessary, IP defragments packets and delivers them to the transport protocol. IP also maintains a database that contains routing information for outgoing packets, which is then utilized to address and fragment the packets before sending them on to the link layer.

The most common link layer protocol utilized for Linux is the Ethernet protocol. The link layer and physical layer work hand in hand. The physical layer is composed of the actual network device, such as Ethernet and token ring. The link layer allows devices in the physical layer to communicate with one another, and thus send information (packets) between them. In the case of our outgoing packet, once the network layer of the new host has been reached, the process is either reversed if the network layer protocol determines the host is the intended destination system of the packet, or forwarded on to another system.

5.4.1.1 Transport layer protocols

The SLES kernel supports two transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

TCP

Transmission Control Protocol is a set of rules used along with the Internet Protocol (IP) to send data between computing systems over the Internet. TCP keeps track of these data messages and organizes them into fragments or packets for delivery, and then reassembles them to the original message upon arrival at the intended remote system. TCP allows for a full-duplex stream of communication between two processes. TCP is a connection-oriented protocol, which means that a connection is established and maintained until such time as the message or messages that are to be exchanged by the application programs at each end have been exchanged. TCP also has the ability to handle flow control. This prevents the source machine from swamping a slower destination machine with data. If the destination machine's buffer becomes full with incoming packets, TCP sends a control signal to the source machine indicating that it cannot handle any more information at the moment and to slow down the transmission. TCP has the ability to handle sequencing. When packets are being sent out, not all of them take the same route, which can result in packets being delivered out of sequence. TCP has a way of reordering the segments to avoid the sender having to resend all the segments. The operation of the TCP protocol can be divided into three distinct sections: establishment of connection, transmission of data, and termination of connection. For more information, see RFC793[1].

UDP

User Datagram Protocol is similar to TCP, but provides fewer error recovery services and features. UDP is primarily used for broadcasting messages over a network. UDP is an unreliable connectionless protocol that is useful for applications that do not require or want TCP's sequencing or flow control. UDP is used for one-shot, request-reply applications where prompt delivery is important. Examples of these types of applications would be DNS (Domain Name System) and transmission of speech or video. For more information, see RFC768.

5.4.1.2 Network layer protocols

The SLES kernel supports two network layer protocols: Internet Protocol (IP) and Internet Control Message Protocol (ICMP).

Internet Protocol

The SLES kernel supports the Internet Protocol version 4. IPv4 is the standard that defines the manner in which network layers of two hosts interact. IPv4 provides a connectionless, unreliable, best-effort packet delivery service. IP stamps a packet with the addresses of the receiver and the sender and determines the proper routing decisions for each packet. A best-effort delivery service means that packets might be discarded during transmission, but not without a good reason.

All IP packets or datagrams consist of a header part and a text part (payload). The payload has a maximum size limit of 65536 bytes per packet. The header also consists of a Time to Live (TTL) that is used to limit the life of the packet on the network. Three fields in the IPv4 header are devoted to fragmentation. If the network layer receives a transport datagram that is too large to transport, it subdivides the data into smaller sized chunks. These fragments are controlled by the DF, MF and Fragment Offset fields in the header.

---

[1] The Requests for Comments (RFCs) form a series of notes, started in 1969, about the Internet (originally the ARPANET). These notes discuss many aspects of computer communication, focusing on networking protocols, procedures, programs, and concepts. For more information see http://isc.faqs.org/rfcs/.

The *DF* stands for Don't Fragment, which is an order to routers not to fragment the datagrams. This bit is set when the destination node is incapable of reassembling the fragments.

The *MF* field stands for More Fragments, which indicates that the current packet does not contain the final fragment in the datagram. Only the final fragment in a series has this bit turned off.

The Fragment Offset field specifies the order in which a particular fragment belongs in the current datagram. All fragments except the last one in a datagram must be a multiple of 8 bytes, which is the elementary fragment unit. Because 13 bits are provided for this field, there is a maximum of 8192 possible fragments per datagram.

The Type of Service field is not commonly used. It allows the host to specify a tradeoff between fast service and reliable service.

Further details on other fields in the IP header can be found in RFC 791.

As discussed previously, every host and router on the Internet has an address that uniquely identifies it and also denotes the network on which it resides. No two machines can have the same IP address. To avoid addressing conflicts, the network numbers have been assigned by the InterNIC (formerly known simply as NIC). Blocks of IP addresses are assigned to individuals or organizations according to one of three categories--Class A, Class B, or Class C. Class D format is used for multicasting, in which a datagram is directed to multiple hosts. Class E format is reserved for future use. These addresses are in the form of 32-bit binary strings. The address is divided into parts that determine the location of a system. The *network* part of the address is common for all machines on a local network. The *host* part of the IP address provides information that is specific to a local network and thus distinguishes one system from another. The host part of an IP address can be further split into a sub-network address and a host address. Subnetworks permit organizations to manage groups of information more effectively.

Internet Control Message Protocol

ICMP is a management protocol and messaging service provider for IP. The primary function is to send messages between network devices regarding the health of the network. ICMP delivers error messages between hosts. ICMP doesn’t use ports to communicate like transport protocols do.

ICMP messages fall into the following three general classes:

- The first class includes various errors that can occur somewhere in the network and that can be reported back to the originator of the packet provoking the error.
- The second class includes gateway-to-host control messages; for example, a source-quench message that reports excessive output and packet loss, and a routing redirect that informs a host that a better route is available for a host or a network via a different gateway.
- The third class includes network address request and reply, network mask request and reply, an echo request and reply, and a timestamp request and reply.

5.4.1.3 Link layer protocols

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

Address Resolution Protocol

On a TCP/IP network, each computer and network device requires a unique IP address and also a unique physical hardware address. Each Network Interface Card (NIC) has a unique physical address that is programmed into the read-only memory chips on the card by the manufacturer. The physical address is also referred to as the Media Access Control (MAC) address. The network layer works with and understands physical addresses, whereas the transport and application layers understand IP addresses. The Address Resolution Protocol maps MAC and IP addresses to convert one into another. When the link layer receives

a frame, ARP broadcasts a frame requesting the MAC address that corresponds to the destination IP address. Each computer on the subnet receives this broadcast frame and all but the computer that has the requested IP address ignore it. The computer that has the destination IP address responds with its MAC address.

5.4.2 Network services interface

The SLES kernel provides a socket interface to programs for obtaining network services from the system. Network services are mainly implemented using the client/server architecture. The following illustrates system calls of the socket interface and how they are used by client and server programs, to establish a communication channel.

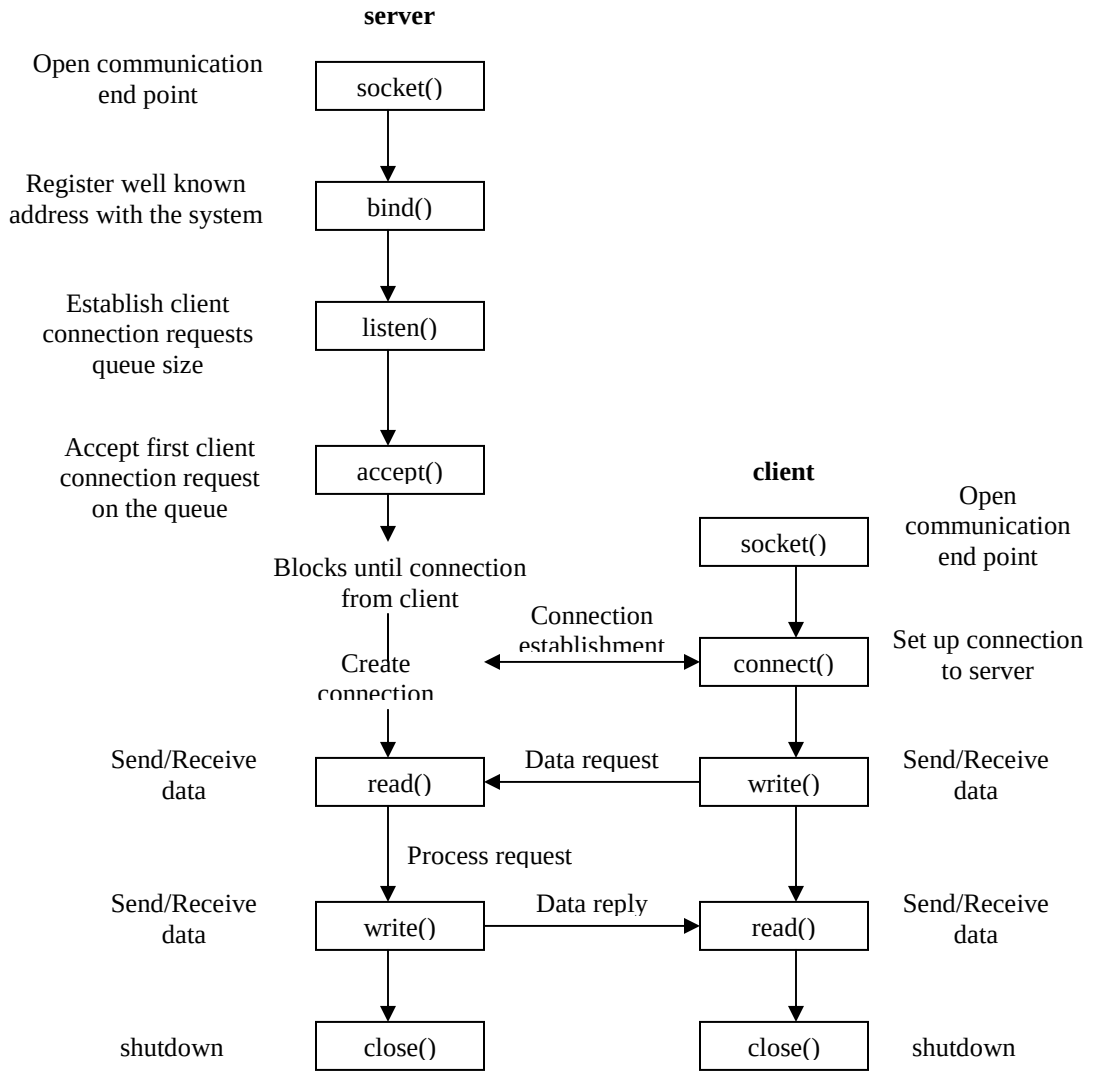

Figure 5-16. Connection establishment using the socket interface

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, using which protocol. A port is a 16-bit number, used by the host-to-host protocol to identify which higher level protocol or application program it must deliver incoming messages to. Sockets, which are described in

section 5.3.5, are communications endpoint. Ports and sockets provide a way to uniquely and uniformly identify connections and the program and hosts that are engaged in them.

The following describes any access control and object reuse handling associated with establishing a communications channel.

`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.

`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.

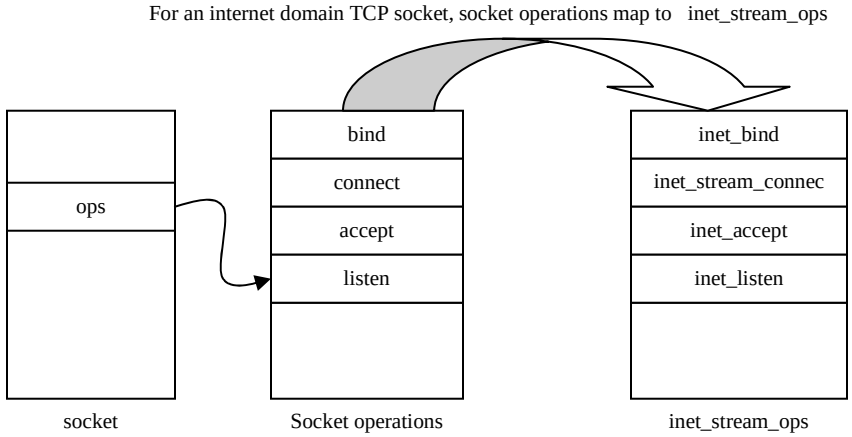

Figure 5-17. bind() function for internet domain TCP socket.

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

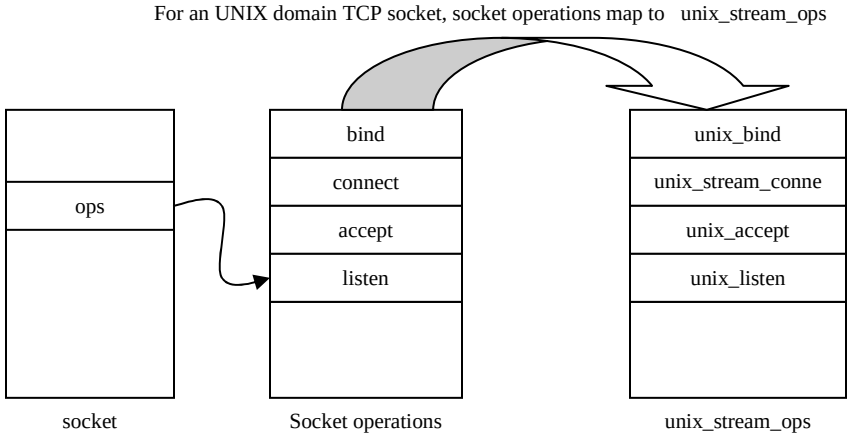

Figure 5-18. bind() function for UNIX domain TCP socket.

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 discretionary access control policies of the ext3 file system described in section 5.1. `bind()` does not create any data objects that are accessible to users and therefore there are no object reuse issues to handle.

`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 any access control. `listen()`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 `listen()` system call.

`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.

`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 any access control. `connect()`does not create any data objects that are accessible to users and therefore there are no object reuse issues to handle.

`read()`, `write()` and `close()`

`read()`, `write()` and `close()` are generic I/O system calls that operate on a file descriptor. Depending on the object type, whether regular file, directory or socket, appropriate object specific functions are invoked. Access control is performed at `bind()` time. `read()`, `write()`, and `close()` operations on sockets do not perform any access control.

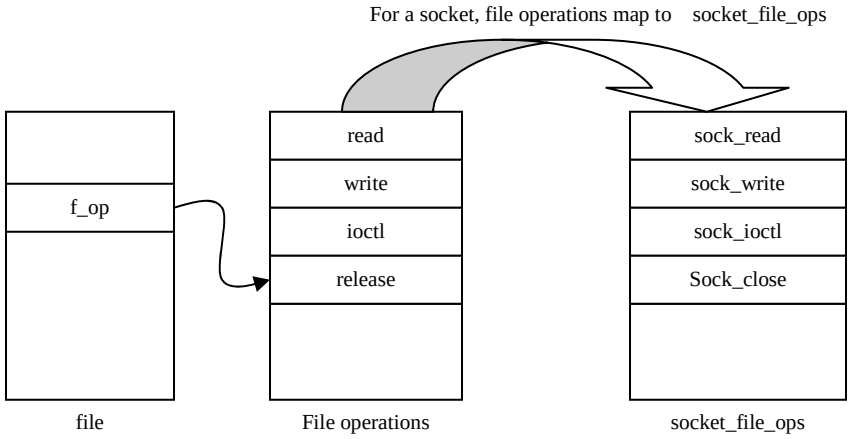

Figure 5-19. Mapping of read, write and close calls for sockets.

5.5 Memory management

This section describes the memory management subsystem of the SLES kernel. The memory management subsystem is responsible for the management of memory resources available on a system. The memory management subsystem includes allocation of physical memory, management of the system and process virtual address spaces, controlled sharing of memory among multiple processes, and allocation of memory for objects implemented in memory. The memory management subsystem interacts with the process subsystem, the network subsystem, the IPC subsystem and the file and I/O subsystem.

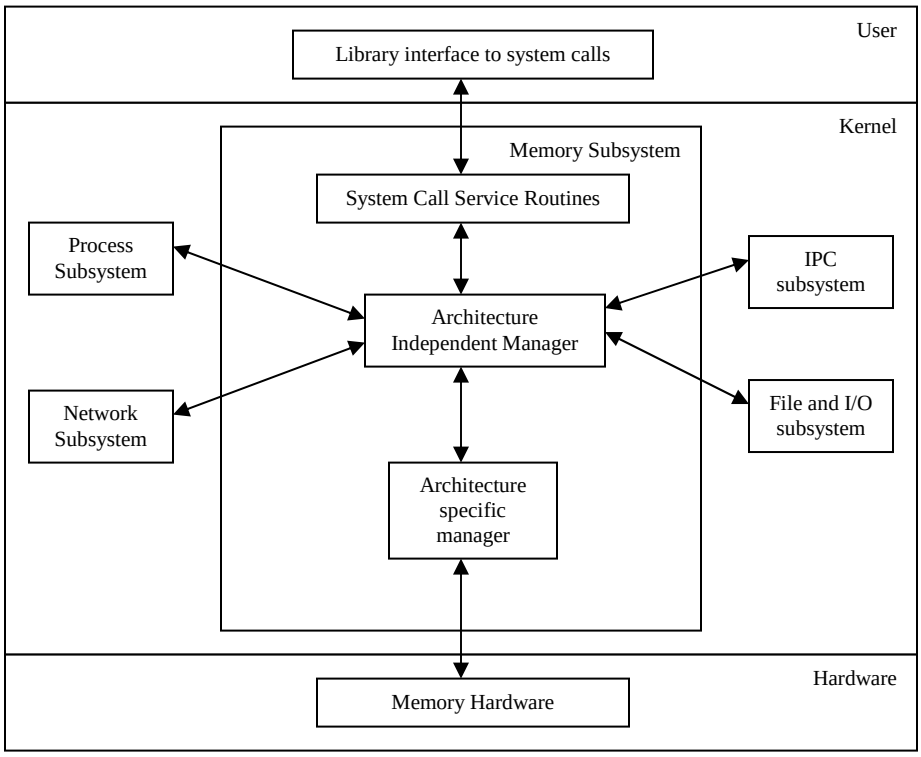

Figure 5-20. Memory subsystem and its interaction with other subsystems

This section highlights the implementation of the System Architecture requirement 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 three subsections. The first 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 second 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 third subsection, “Process Address Space,” describes how a process views dynamic memory and what the different components are of a process’s address space. The third subsection also highlights how the kernel enforces access control with memory regions and handles object reuse with demand paging.

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 xSeries, pSeries, iSeries, zSeries, and eServer 325 lines of servers, which are all part of the TOE.

5.5.1 Memory addressing

A memory address provides a way to access the contents of a memory cell. 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 following sections describe memory addressing for xSeries, pSeries, iSeries, zSeries, and eServer 325 systems.

5.5.1.1 xSeries

The following briefly describes the xSeries memory addressing scheme. For more detailed information on the xSeries memory management subsystem, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

On xSeries computers, there are three address types:

Logical address
: The address is included in the machine language instruction for an operand or for an instruction. A logical address consists of a segment and an offset that signifies the distance from the start of the actual address.

Linear address
: A 32-bit unsigned integer that can address up to 4,249,967,296 (4GB) memory cells.

Physical address
: A 32-bit unsigned integer that addresses memory cells in physical memory chips.

To access a particular memory location, the CPU, using its segmentation unit, transforms a logical address into a linear address, which in turn is translated into a physical address by the CPU paging unit.

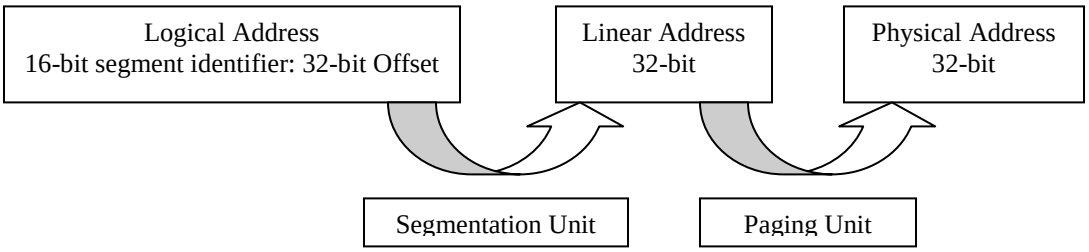

Figure 5-21. xSeries address types and their conversion units

5.5.1.1.1 Segmentation

The segmentation unit translates a logical address into a linear address. A logical address has two parts: A 16-bit segment identifier called the segment selector, and a 32-bit offset. For fast retrieval of the segment selector, the processor provides six segmentation registers to hold segment selectors. Each segmentation register has a specific purpose. For example, the code segment (cs) register points to a memory segment that contains program instructions. The code segment 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 characteristics of the 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.

The following figure schematically describes access control as enforced by memory segmentation.

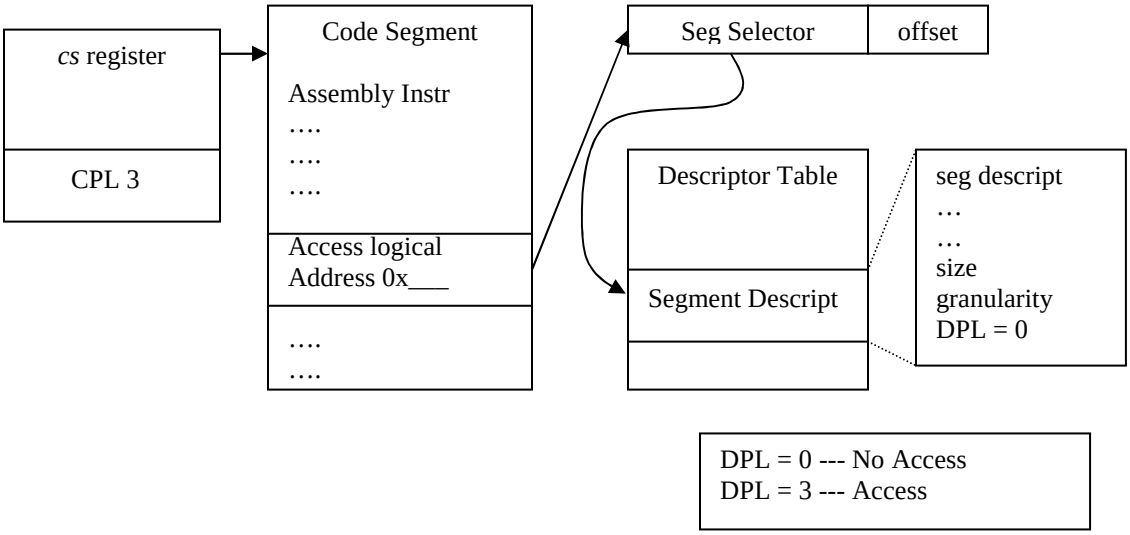

Figure 5-22. Access control through segmentation

5.5.1.1.2 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.

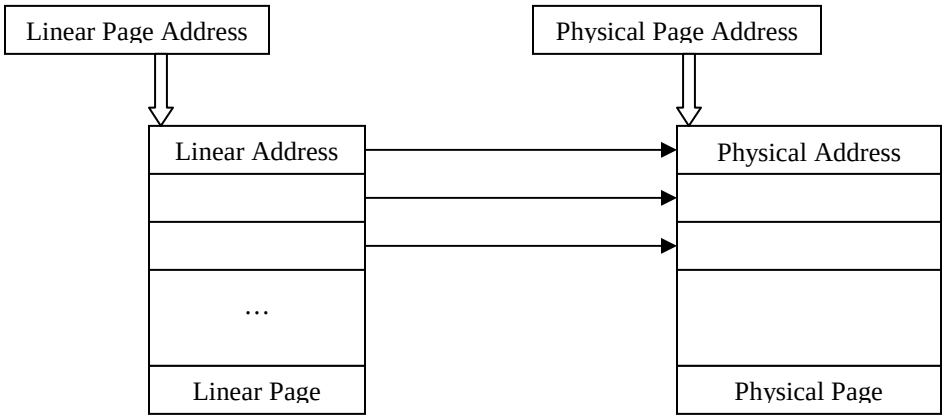

Figure 5-23. Contiguous linear addresses map to contiguous physical addresses

The paging unit sees all Random Access Memory 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.

5.5.1.1.2.1 Paging in hardware

The xSeries 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 location representations for the following:

Directory
The most significant 10-bits.

Table
The intermediate 10-bits.

Offset
The least significant 12-bits.

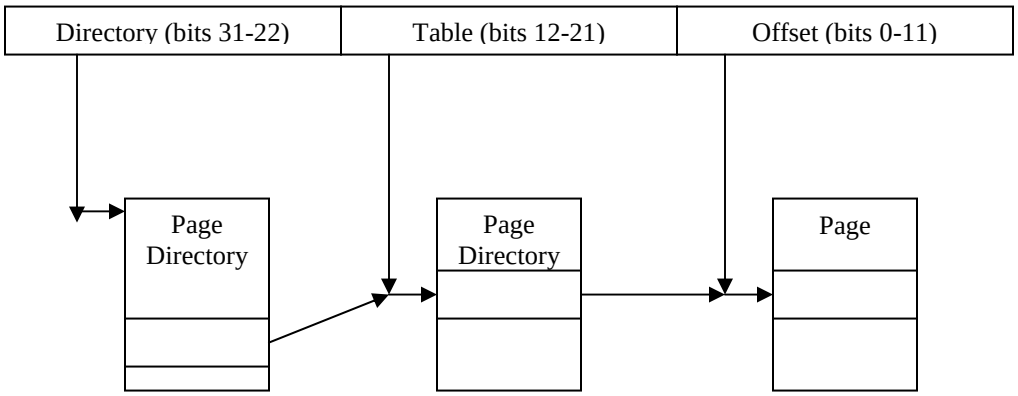

Figure 5-24. 32 bit linear address

In extended paging, 32-bits of linear address are divided into location representations for the following:

Directory
The most significant 10-bits.

Offset
The remaining 22-bits.

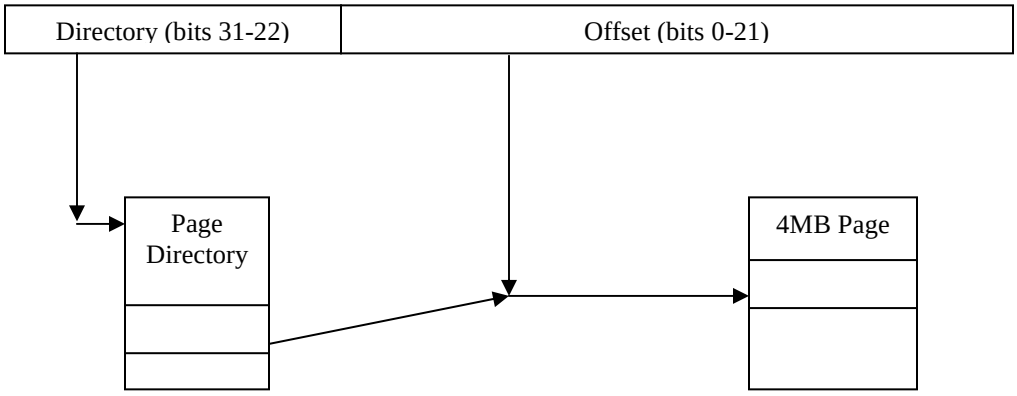

Figure 5-25. 32 bit linear address – extended paging

Each entry of the page directory and 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

Contains the 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

Contains the privilege level that is required to access 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 it can be accessed always.

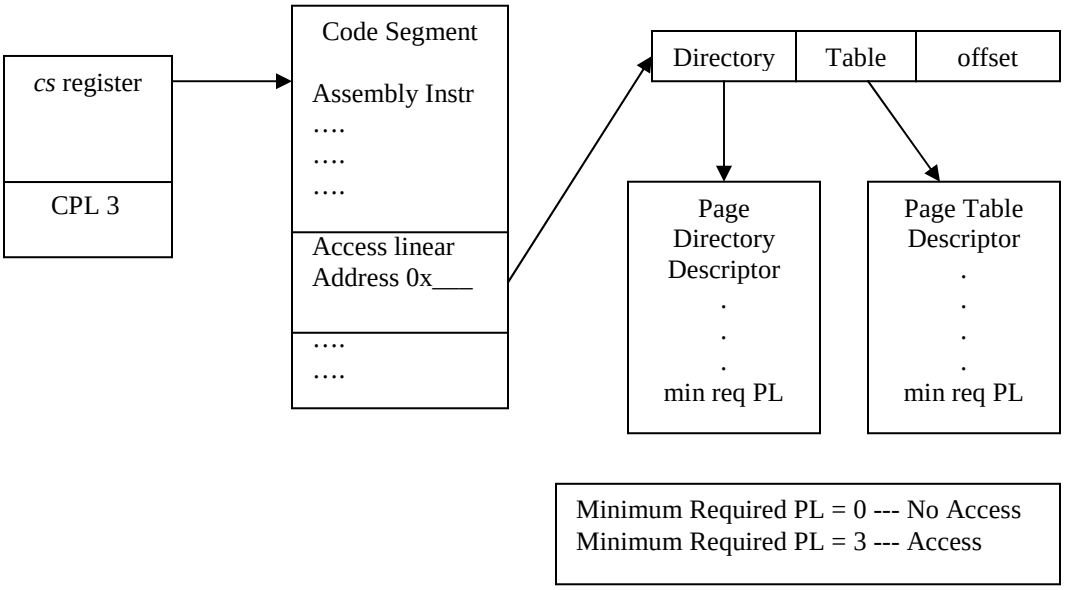

Figure 5-26. Access control through paging

5.5.1.1.2.2 Paging in the SLES kernel

The SLES kernel is version 2.6.5 of Linux. The SLES kernel 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 xSeries systems, the page middle directory field is eliminated when it is set to zero.

5.5.1.1.2.3 Access control for control transfers through call gates

Intel processors 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. 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.

5.5.1.1.3 Translation Lookaside Buffers (TLB)

The xSeries 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 5.5.1.1.2, 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 5.5.1.1.2.

5.5.1.2 pSeries

Linux on pSeries systems can run in native mode or in a logical partition. Memory addressing for the SLES kernel running in logical partition is covered in the iSeries section 5.5.1.3. Both iSeries and pSeries use either POWER4 or POWER5 processors. POWER5 processor based systems only support SLES kernel running in a logical partition. This section describes the pSeries memory addressing for the SLES kernel running in native mode. For more detailed information on the pSeries memory management subsystem, please refer to the following:

Engebretsen David, *PowerPC 64-bit Kernel Internals*,
http://oss.software.ibm.com/linux/presentations/ppc64/ols2001/ppc64-ols-2001.ps

pSeries hardware documents at *http://www.ibm.com/eserver/pseries*

On pSeries systems, there are four address types:

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).

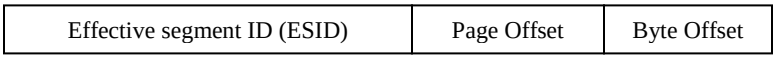

Figure 5-27. Effective address

Virtual address

The virtual address, which is equivalent to the linear address of xSeries, 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.

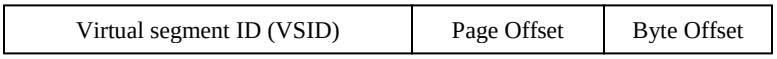

Figure 5-28. Virtual address

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-Kbyte to 256-Mbyte. The block address is the effective address of a block.

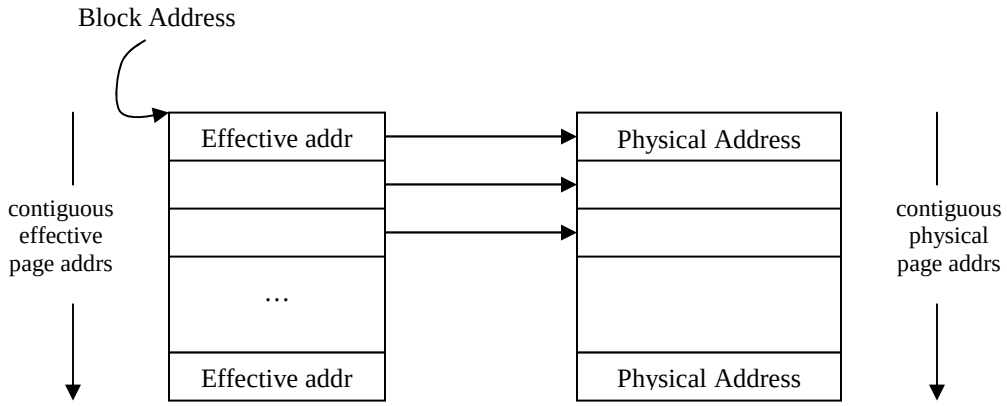

Figure 5-29. Block address

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 size 128-Kbyte to 256-Mbyte.
- 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.

Machine State Register (MSR)

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

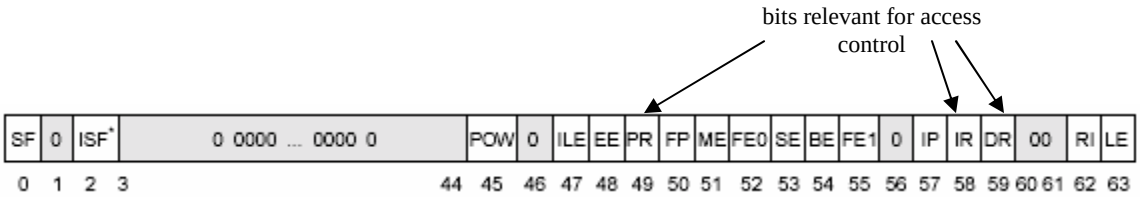

Figure 5-30. Machine State Register

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.

Page descriptor
Pages are described by Page Table Entries (PTEs). PTEs are generated and placed in a page table in memory by the operating system. 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.

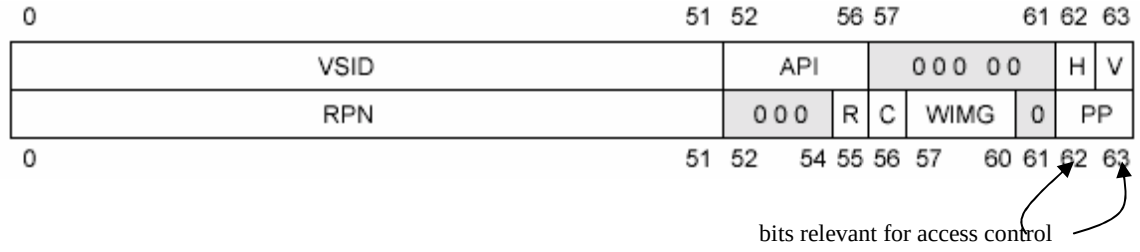

Figure 5-31. Page Table Entry

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

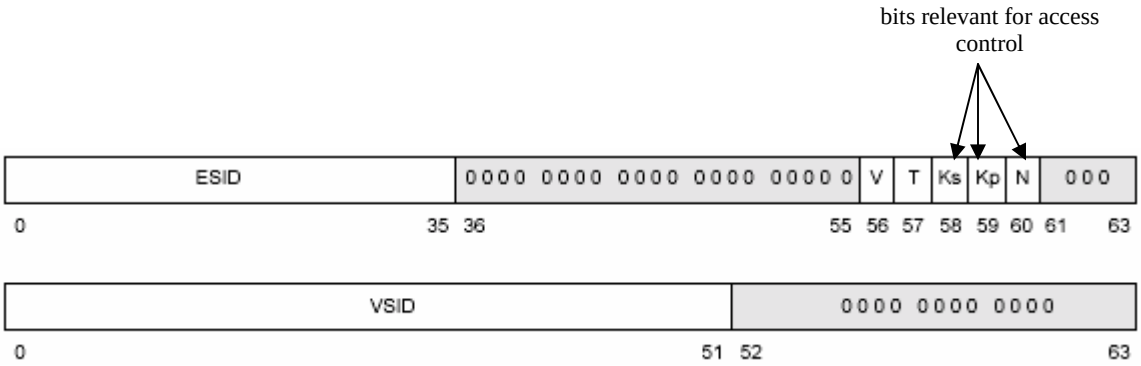

Figure 5-32. Segment Table Entry

Ks – Supervisor-state protection key
Kp – User-state protection key
N – No-execute protection bit

Block descriptor
For address translation, each block is defined by a pair of special purpose registers called upper and lower BAT (Block Address Translation) registers that contain effective and physical addresses for the block.

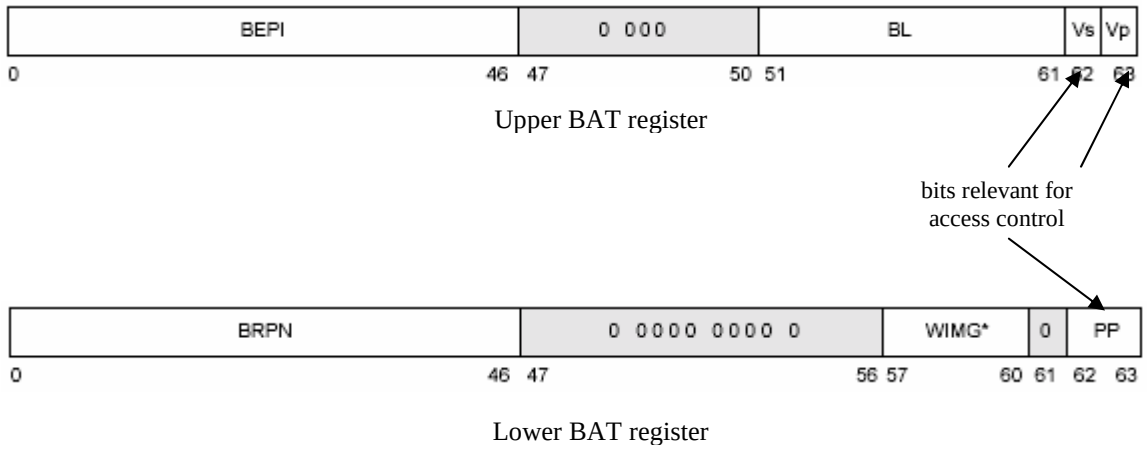

Figure 5-33. Block Address Translation entry

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.

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.

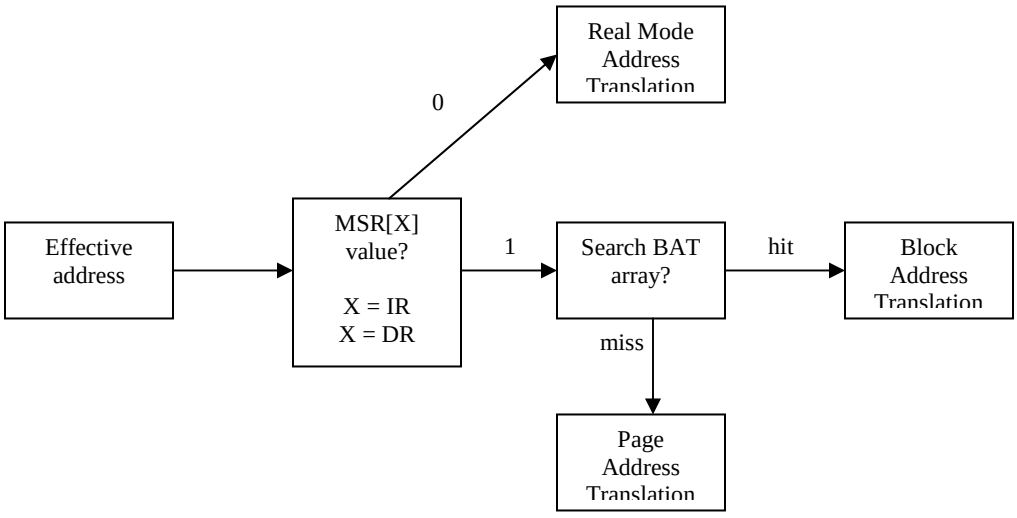

Figure 5-34. Address translation method selection

The following sections describe the three address translation mechanisms and the access control performed by them.

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. This mode is used by the operating system during initialization and some interrupt processing. Because there is no translation, there is no access control implemented for this mode. However, because only the superuser can alter MSR[IR] and MSR[DR], there is no violation of security policy.

Block Address Translation and access control
Block Address Translation checks to see if the effective address is within a block defined by the BAT array. If it is, Block Address Translation goes through the steps described in Figure 5-21 to perform the access check for the block and get its physical address.

Block Address Translation 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, Block Address Translation 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 "no-execution" bit on a per-segment basis, allowing the Page Address Translation mechanism to implement access control based on instruction or data fetches.

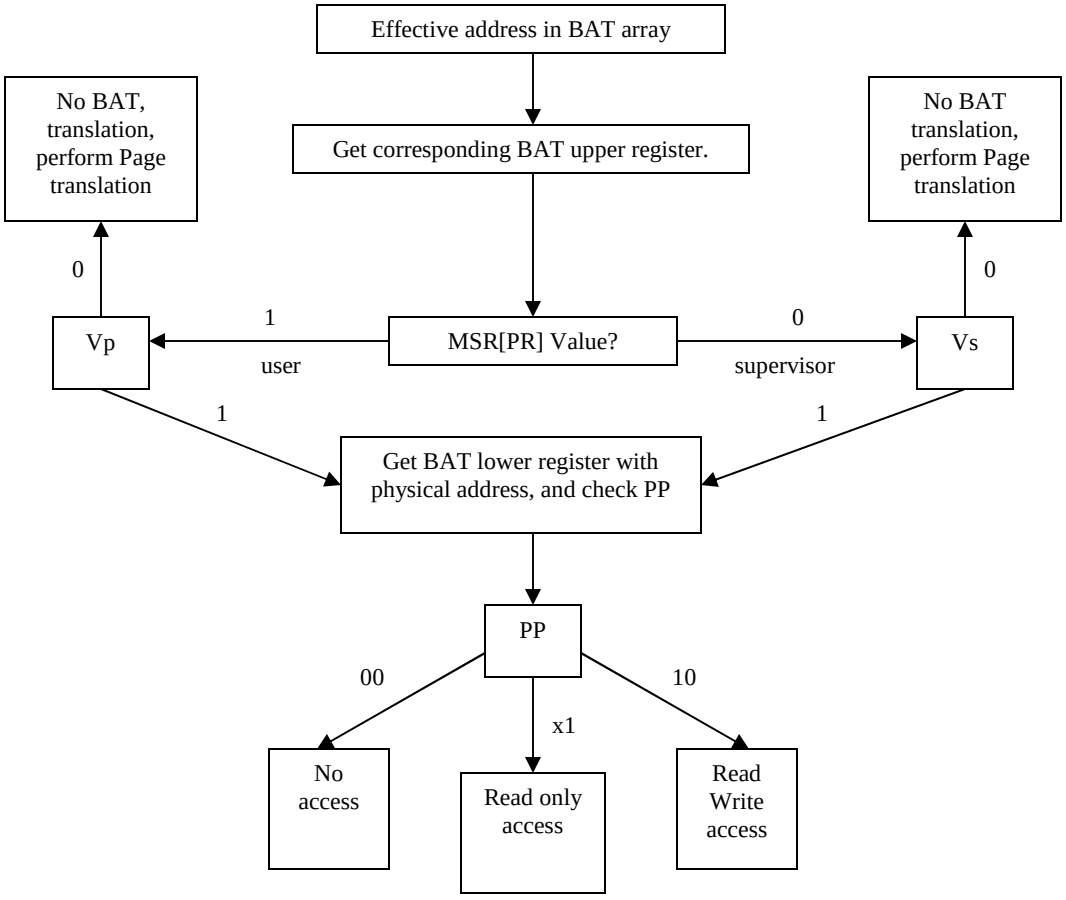

Figure 5-35. Block Address Translation access control

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.

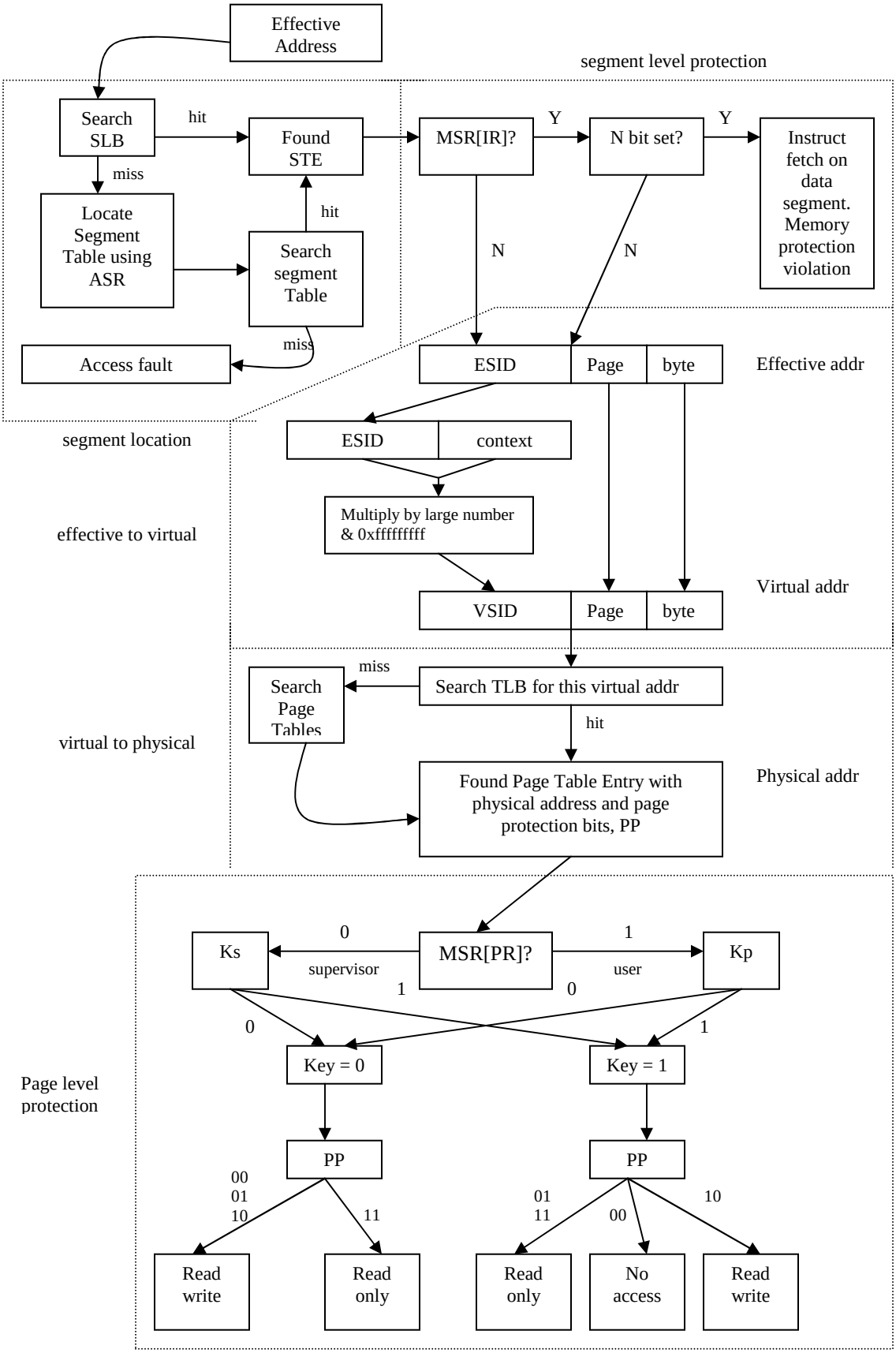

Figure 5-36. Page Address Translation and access control

5.5.1.3 iSeries

iSeries hardware uses 64-bit PowerPC processors. iSeries differs from pSeries in its I/O architecture (however, they both use the same set of processors). Both iSeries and pSeries systems support Logical Partitions (LPAR). Logical partitions divide the hardware resources, such as processors, memory, and portions of real storage, to create multiple "logical" systems that can run their own copy of an operating system. Unlike pSeries systems, which support running SLES in either direct native mode or in a logical partition, SLES can run only in logical partitions on iSeries systems. This section describes logical partitions and their impact on memory addressing and access control. For detailed information on LPAR please refer to the following:

Engebretsen David, PowerPC 64-bit Kernel Internals,
http://oss.software.ibm.com/linux/presentations/ppc64/ols2001/ppc64-ols-2001.ps

David Boutcher, The Linux Kernel on iSeries, *http://www.ibm.com/iseries/linux*

iSeries hardware documents at *http://www.ibm.com/eserver/iseries*

OS/400 V5R3 documents at *http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm* & *http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/rzaq9.pdf*

The number of partition an iSeries can be partitioned into depends on the processor. POWER5 based iSeries systems running V5 R3 can be portioned in to as many as 254 partitions. Partitions share processors and memory. A partition can be assigned processors in increments of 0.01. The figure below represents a 4 CPU system that is split into 4 logical partitions. The primary partition runs OS/400 V5R2 or OS/400 V5R3 (i5/OS). From the perspective of the TOE Security Functions, V5R2 and V5R3 are equivalent. Primary partition is assigned 0.25 of a CPU, while other partitions running SLES 9 are assigned 2, 1 and 0.75 CPUs, respectively. The hypervisor provides pre-emptive timeslicing 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.

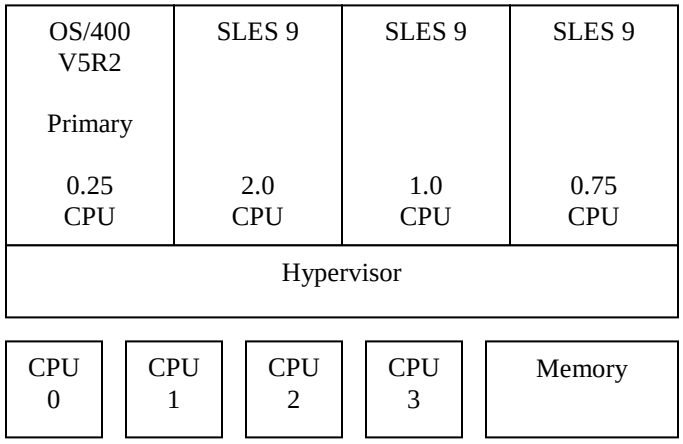

Figure 5-37. Logical partitions

On pSeries 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 both pSeries and iSeries necessitate a 3rd mode of operation for the processor. This 3rd mode, called the hypervisor mode, 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 Machine State Register.

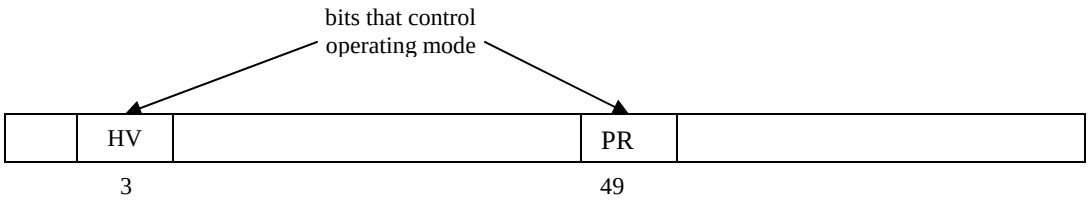

Figure 5-38. Machine State Register

PR – Privilege Level. The Privilege Level takes the value of 0 for the supervisor level and 1 for the user level.
HV – Hypervisor. The hypervisor takes the value of 1 for hypervisor mode and 0 for user and supervisor mode.

The following diagram describes the process that determines the operating mode of the processor based on MSR[PR] and MSR[HV] values.

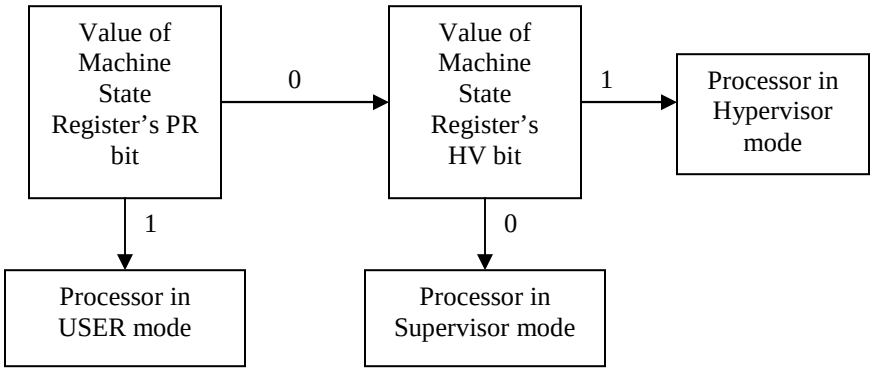

Figure 5-39. Determination of processor mode in LPAR

Just as certain memory areas are protected from access in user mode, there are memory areas, such as hardware page tables, that are accessible only in hypervisor mode. The PowerPC 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.

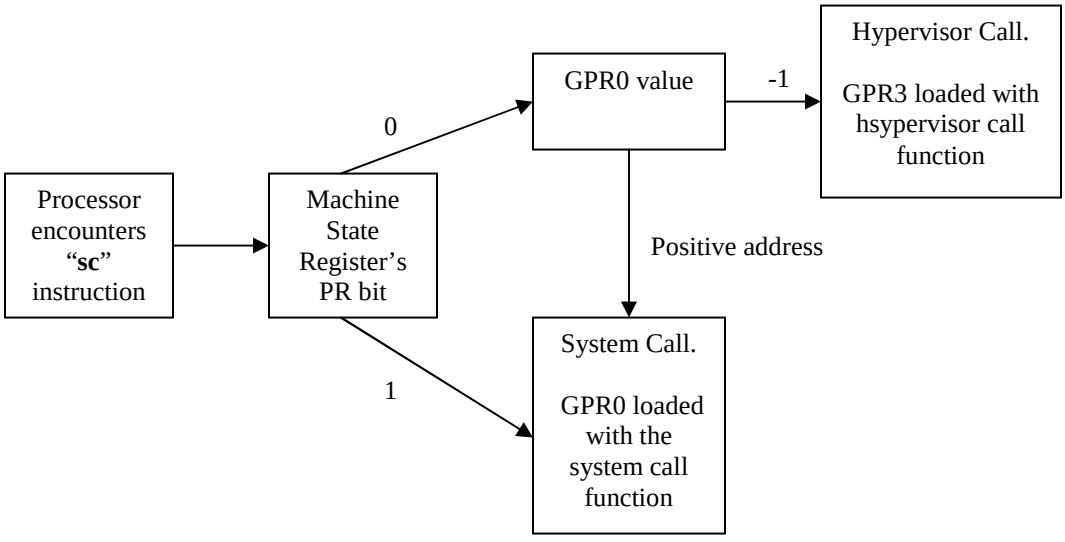

Figure 5-40. Transition to supervisor or hypervisor state

Actual physical memory is shared between logical partitions. Therefore, one more level of translation beyond the four levels described by pSeries section 5.5.1.2 is needed 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. Because iSeries uses the same PowerPC processor described in the pSeries section 5.5.1.2, the iSeries mechanism for translating effective-to-virtual and virtual-to-physical is identical to that of the native pSeries. The only addition is the physical-to-absolute address translated by the hypervisor, as illustrated below.

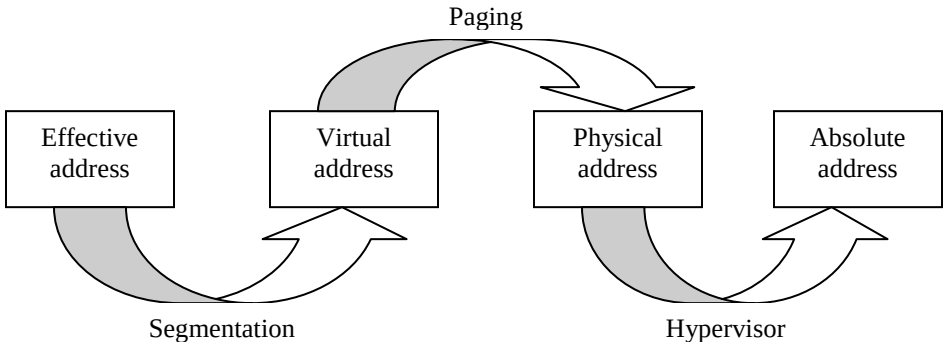

Figure 5-41. Address translation in LPAR

On iSeries and pSeries systems running with logical partitions, the effective address, the virtual address, and the physical address format and meaning are identical to those of pSeries systems running in native mode. The kernel creates and translates them from one another using the same mechanisms described in section 5.5.1.2. Access control by Block Address Translation and Page Address Translation, described in section 5.5.1.2, is performed here as well. The Block Address Translation and Page Address Translation mechanisms provide iSeries and pSeries 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 pSeries native mode and iSeries and pSeries LPAR mode.

The difference between pSeries native mode and iSeries and pSeries LPAR mode comes from the kernel's logical view of the physical addresses versus the absolute memory addresses used by the processor. The LPAR-specific code splits memory into 256-Kbyte "chunks." To map these "chunks" to physical addresses expected by the kernel memory model, the iSeries hypervisor code builds a translation table, called `msChunks` array, to translate physical addresses to absolute addresses. The `msChunks` array is indexed by (logical_address >> 18) and provides a translation from the logical address (kernel logical view of the address, i.e., physical address) to the absolute address, as follows:

Absolute address = (msChunks[logical_address>>18]<<18) | (logical_address&0x3ffff)

The kernel is not aware that the physical address is not the final address used by the processor to access a memory cell. When the kernel attempts an access using the physical address, the hypervisor intercepts the access and converts the physical address to the absolute address using the `msChunks` array. This process allows it to appear to the kernel that there is contiguous memory starting at physical address zero, while in fact the absolute addresses are not contiguous, as illustrated in Figure 5-52.

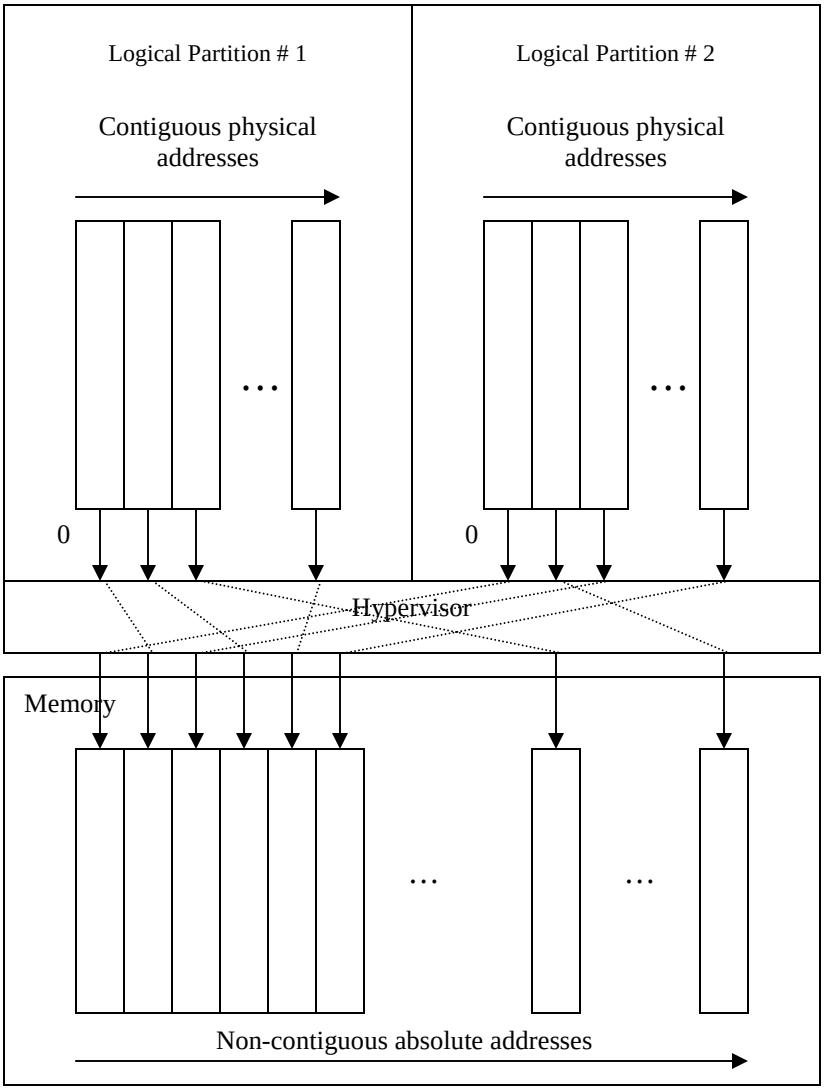

Figure 5-42. Absolute addresses

The hypervisor interacts with the operating system in the logical partition through two data structures that are used to store systemwide and processor specific information. The first data structure, the `naca` (node address communications area), is used to hold systemwide information, such as the number of processors in the system or partition, the size of real memory available to the kernel, and cache characteristics. The `naca` also contains a field used by the iSeries hypervisor, which is initialized to point to the data area used by the hypervisor to communicate system configuration data to the kernel. The `naca` is located at the fixed real address of 0x4000. The second data structure, the `paca` (processor address communications area), contains information pertaining to each processor. Depending on the number of processors, an array of `paca` structures is created.

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 a system call, `rtas`, 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. `rtas` ensures that the calling process posseses the `CAP_SYS_ADMIN` capability.

5.5.1.4 zSeries

There are three common alternatives for running SLES on zSeries systems. SLES can run on native hardware, in Logical Partitions (LPAR), and as z/VM® guests. This section briefly describes these three modes and how they address and protect memory. For more detailed information on zSeries architecture, please refer to the following:

z/Architecture Principle of Operation, http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf
zSeries hardware documents at *http://www.ibm.com/eserver/zseries*

Native Hardware mode
In native hardware mode, SLES runs directly on zSeries hardware. 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.

Logical Partition Mode (LPAR)
In logical partition mode, zSeries 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 pre-emptive timeslicing 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. 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.

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 to a guest operating system the same interface definition provided by the real hardware. Guests operate independent of each other even though they share memory, processors, and devices. z/VM Guest mode provides even more flexibility than LPAR mode because, unlike logical partitions, z/VM virtual machines allow dynamic addition or deletion of memory and devices. 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 zSeries hardware. The Control Program (either LPAR or VM or both) sets up their paging tables and zoning array so that the SIE (Start Interpretive Execution) instruction can do the address conversion. The control program doesn't actively convert any addresses.

5.5.1.4.1 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 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 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.

5.5.1.4.2 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, the addressing mode is 24-bit. If they are 0 and 1, the addressing mode is 31-bit. If they are both 1, 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.

5.5.1.4.3 Address spaces

An address space is a consecutive sequence of integer numbers (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, namely 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 memory area that transfers data between kernel and user space. The secondary address space is not used on 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.

5.5.1.4.4 Address translations

Address translation on z/Architecture can involve two steps. The first one, 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 one 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.

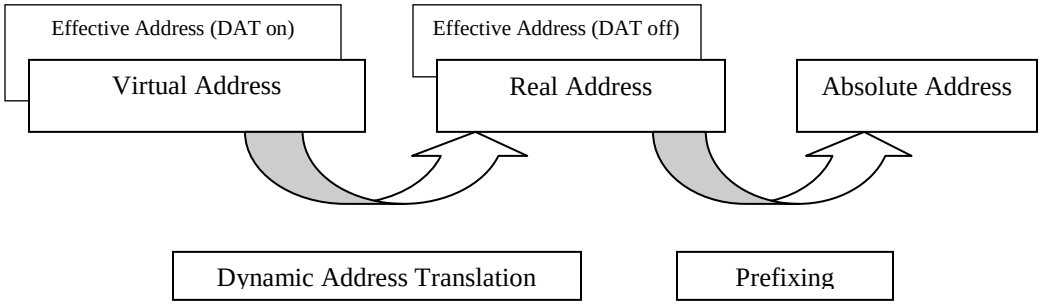

Figure 5-43. zSeries address types and their translation

Dynamic address translation

Bit 5 of the current Program Status Word 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.

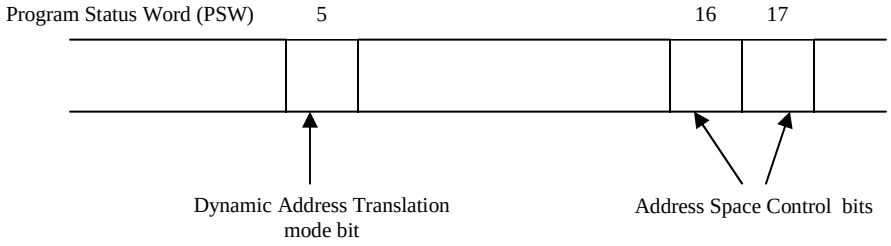

Figure 5-44. Program Status Word

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).

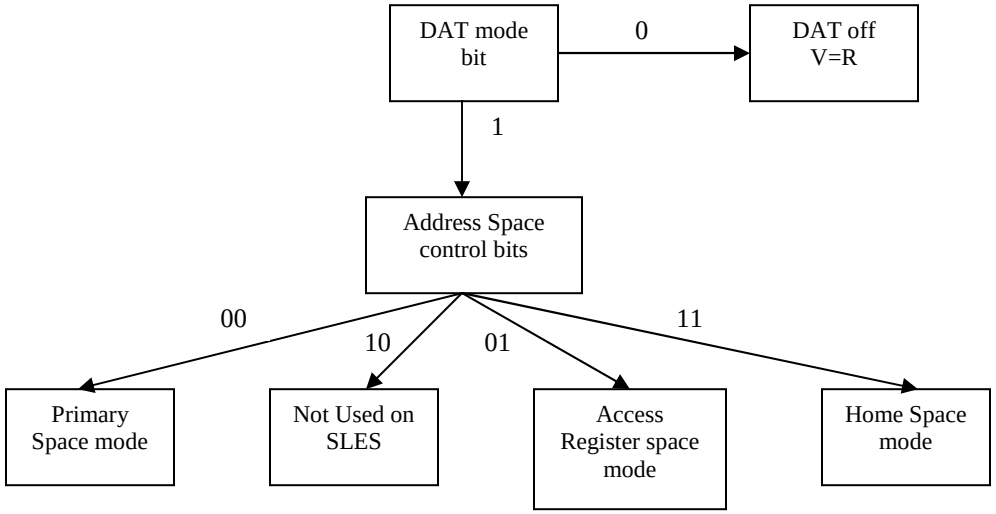

Figure 5-45. Address translation modes

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

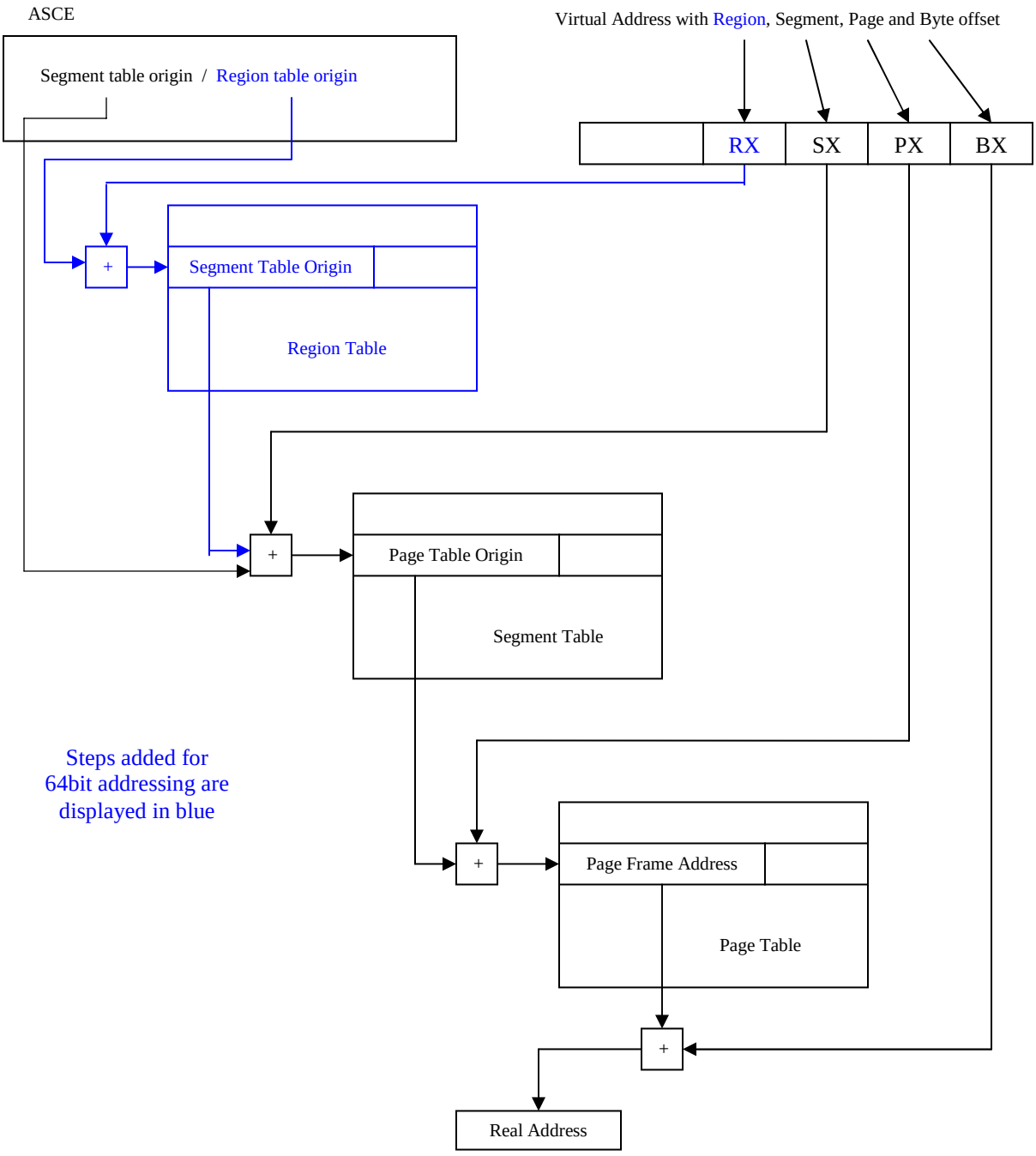

Figure 5-46. 64-bit or 31-bit Dynamic Address Translation

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, please refer to the following:

z/Architecture Principle of Operation, http://publibz.boulder.ibm.com/epubs/pdf/dz9zr002.pdf

5.5.1.4.5 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 Program Status Word (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.

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 4K-byte address blocks).

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

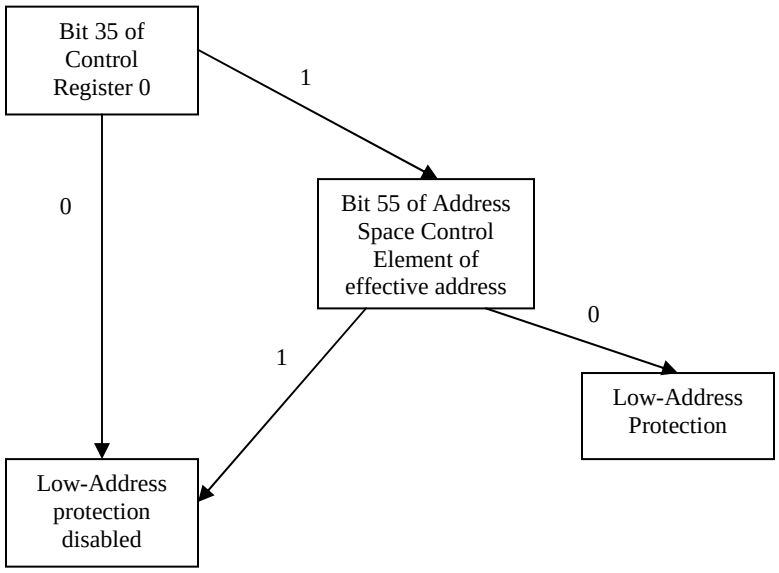

Figure 5-47. Low-address protection on effective address

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.

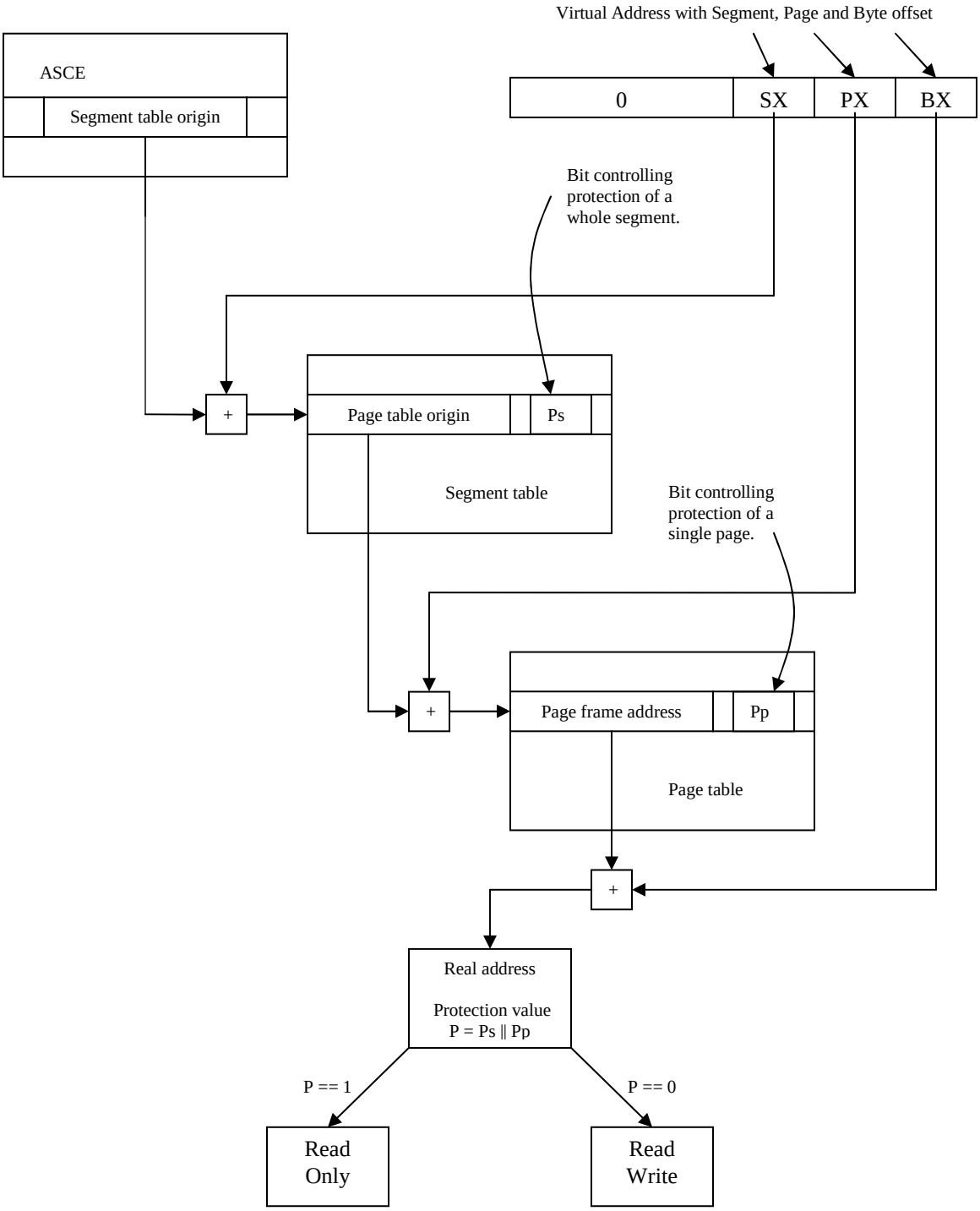

Figure 5-48. 31-bit Dynamic Address Translation with page table protection

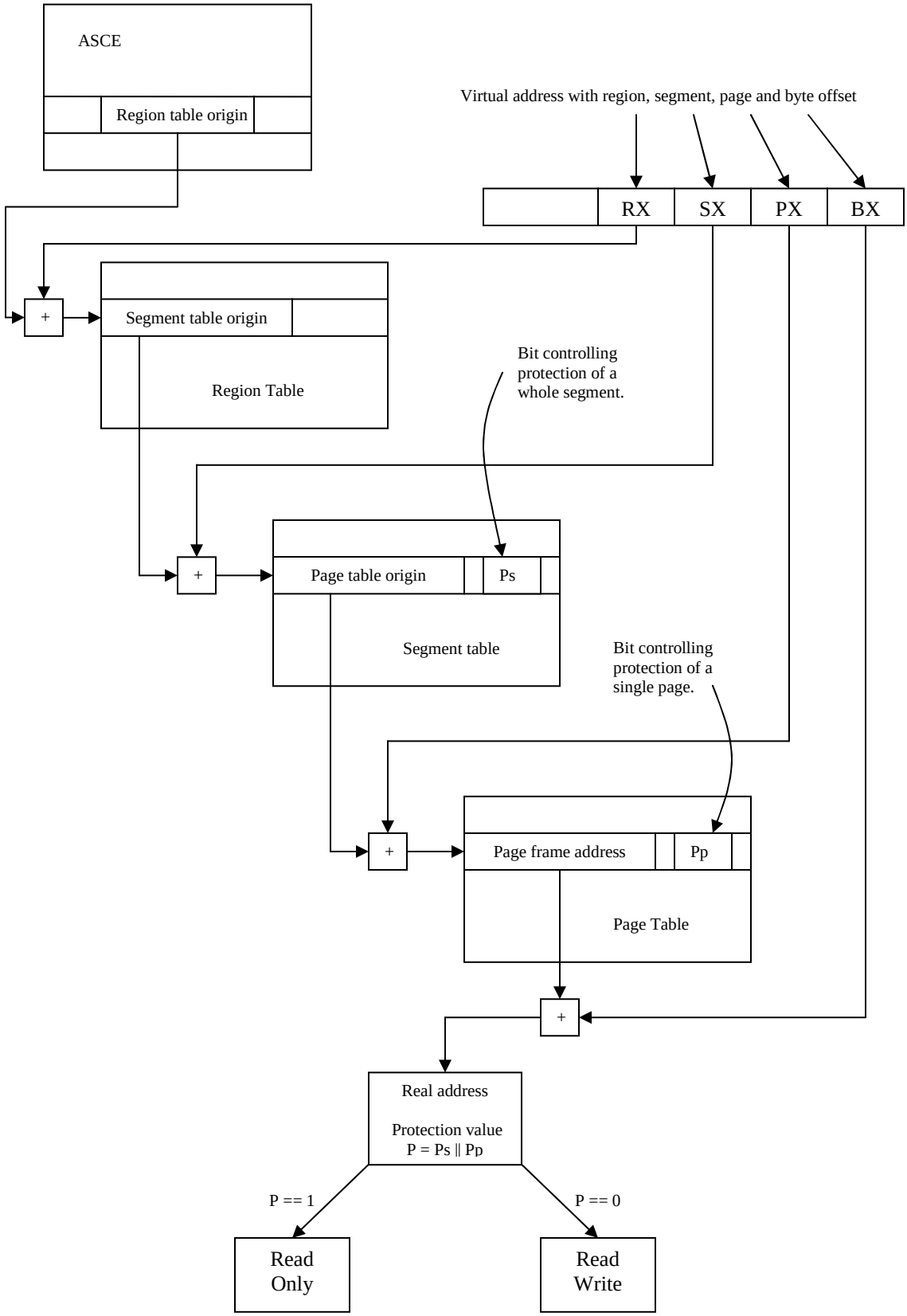

Figure 5-49. 64-bit Dynamic Address Translation with page table protection

Key-controlled protection
Key-controlled protection is applied when an access attempt is made for an absolute address, which refers to a memory location. 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, 3), fetch protection bit (4), reference bit (5), and change bit (6).

Figures 5-50 and 5-51 describe the key-controlled protection.

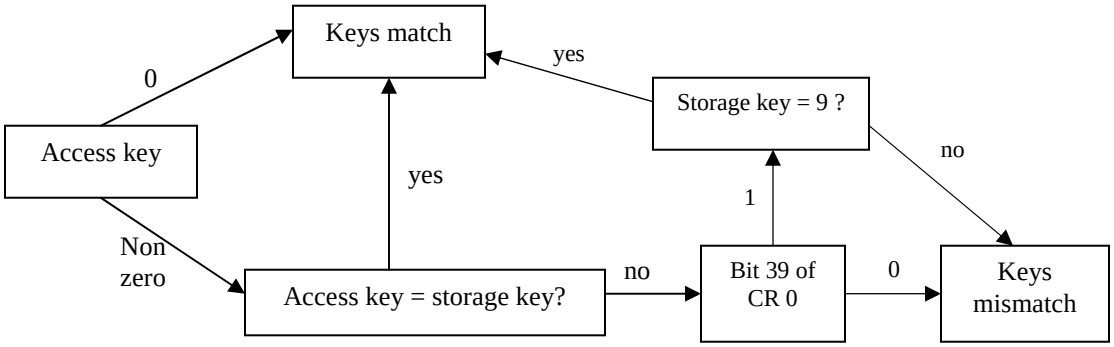

Figure 5-50. Key match logic for key-controlled protection

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.

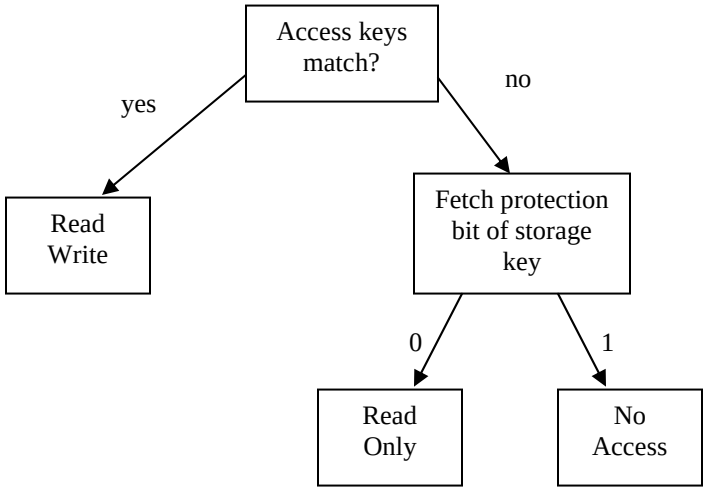

Figure 5-51. Fetch protection override for key-controlled

5.5.1.5 eServer 325

The following briefly describes the eServer 325 memory addressing scheme. For more detailed information on the eServer 325 memory management subsystem, please refer to the following:

AMD64 Architecture, Programmer's Manual Volume 2: System Programming,
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

SUSE Linux Enterprise Server 8 for AMD64,
http://www.suse.com/en/business/products/server/sles/misc/sles8_amd64.pdf

Andi Kleen, Porting Linux to x86-64, *http://old.lwn.net/2001/features/OLS/pdf/pdf/x86-64.pdf*

eServer 325 systems are powered by AMD Opteron processors. The Opteron processor 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 achtitecture described in the xSeries sections of this document. SLES on eServer 325 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.

On eServer 325 computers, there are the following four address types:

Logical address
The address is included in the machine language instruction for an operand or for an instruction. A logical address consists of a segment selector and the effective address. The segment selector specifies an entry in either the global or local descriptor table. The effective address is an offset that signifies the distance from the start of the segment specified by the segment selector.

Effective Address
The effective address is the offset into a memory segement. Long mode supports 64-bit effective address length.

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

Physical address
The physical address is a reference into the physical address space. Physical address is the address assigned to a main memory location. Physical addresses are translated from virtual addresses using the paging mechanism. On eServer 325 systems, 40-bit physical addresses allow access to 1 Terabyte of physical address space.

To access a particular memory location, the CPU, using its segmentation unit, transforms a logical address into a linear address, which in turn is translated into a physical address by the CPU paging unit.

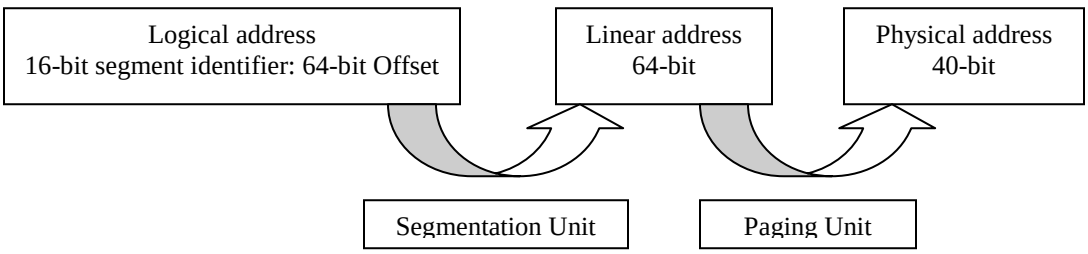

Figure 5-52. eServer 325 address types and their conversion units

Access control and protection mechanisms are part of both segmentation and paging. The following sections describe how segmentation and paging are used to provide access control and memory resource separation by SLES on IBM eServer 325 systems.

5.5.1.5.1 Segmentation

The segmentation unit translates a logical address into a linear address. A logical address has two parts: A 16-bit segment identifier called the segment selector, and a 64-bit offset. For fast retrieval of the segment selector, the processor provides six segmentation registers to hold segment selectors. Each segmentation register has a specific purpose. For example, the code segment (cs) register points to a memory segment that contains program instructions.

Each segment is represented by a segment descriptor that describes characteristics of the 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 Global Descriptor Table Register (GDTR), while the LDT is accessed through the Local Descriptor Table Register (LDTR). 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 protection is used to isolate memory resources belonging to one process from that of another. The segment protection mechanism uses the concept of privilege levels 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 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.

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 (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 diagrams in Figure 5-53 illustrate data-access privilege checks.

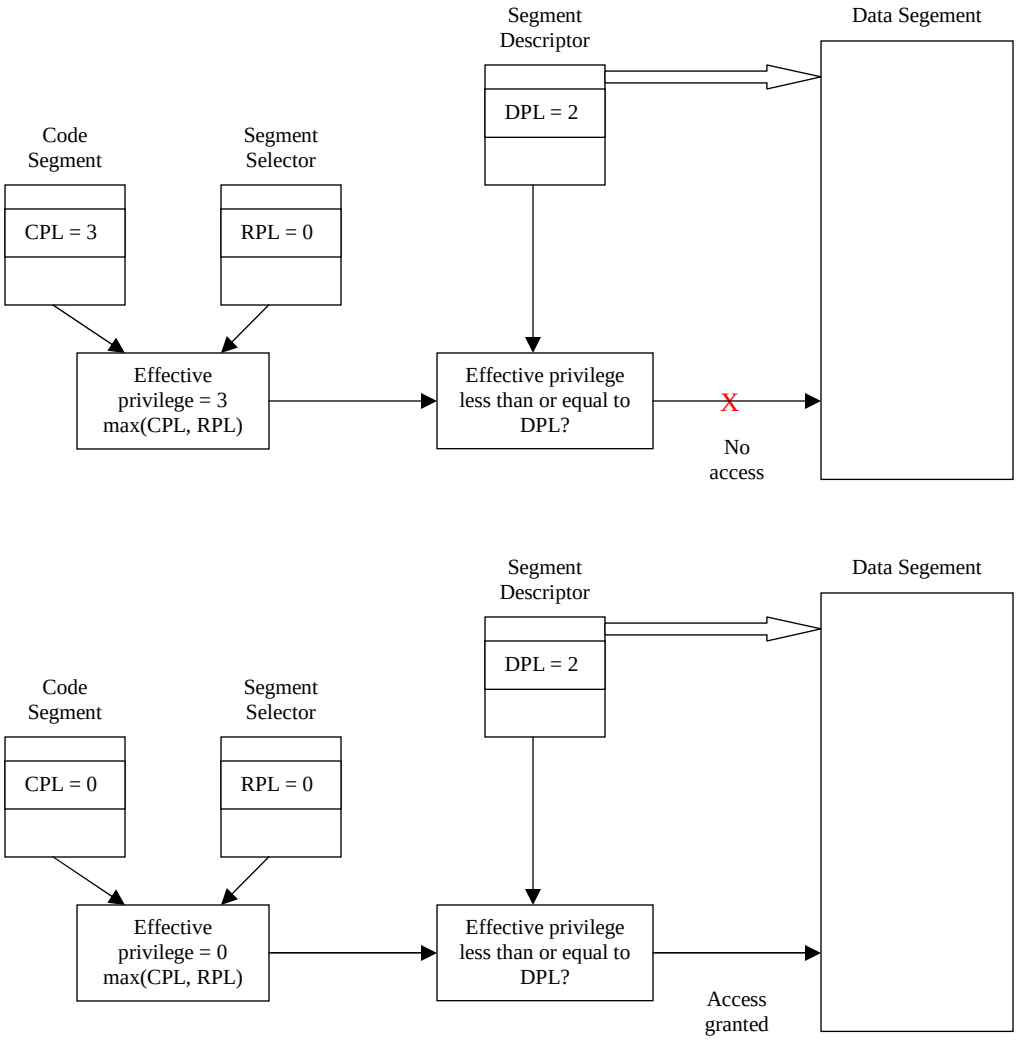

Figure 5-53. Data access privilege checks

Access control for stack segments

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

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 specific instructions `SYSCALL` and `SYSRET` to perform system calls. If the code segment is non-conforming (conforming bit "C" set to zero in segment descriptor), the processor first checks to ensure that CPL is equal to DPL. If CPL is equal to DPL, 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 (conforming bit "C" set to one in the segment descriptor), 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, access is allowed. Otherwise, a general protection exception occurs. RPL is ignored for conforming segments.

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. 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.

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.

5.5.1.5.2 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.

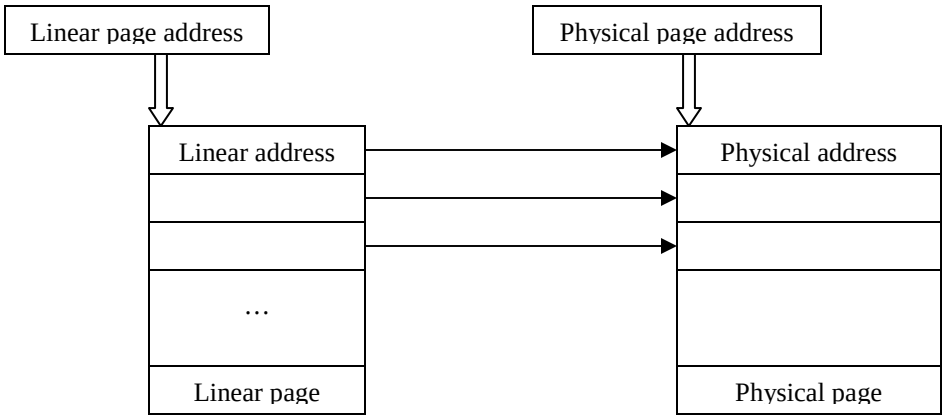

Figure 5-54. Contiguous linear addresses map to contiguous physical addresses

The paging unit sees all Random Access Memory 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 eServer 325 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 K-byte and 2 M-byte.

The following figure illustrates how paging is used to translate a 64-bit linear address is into a physical address for the 4 K-byte page size.

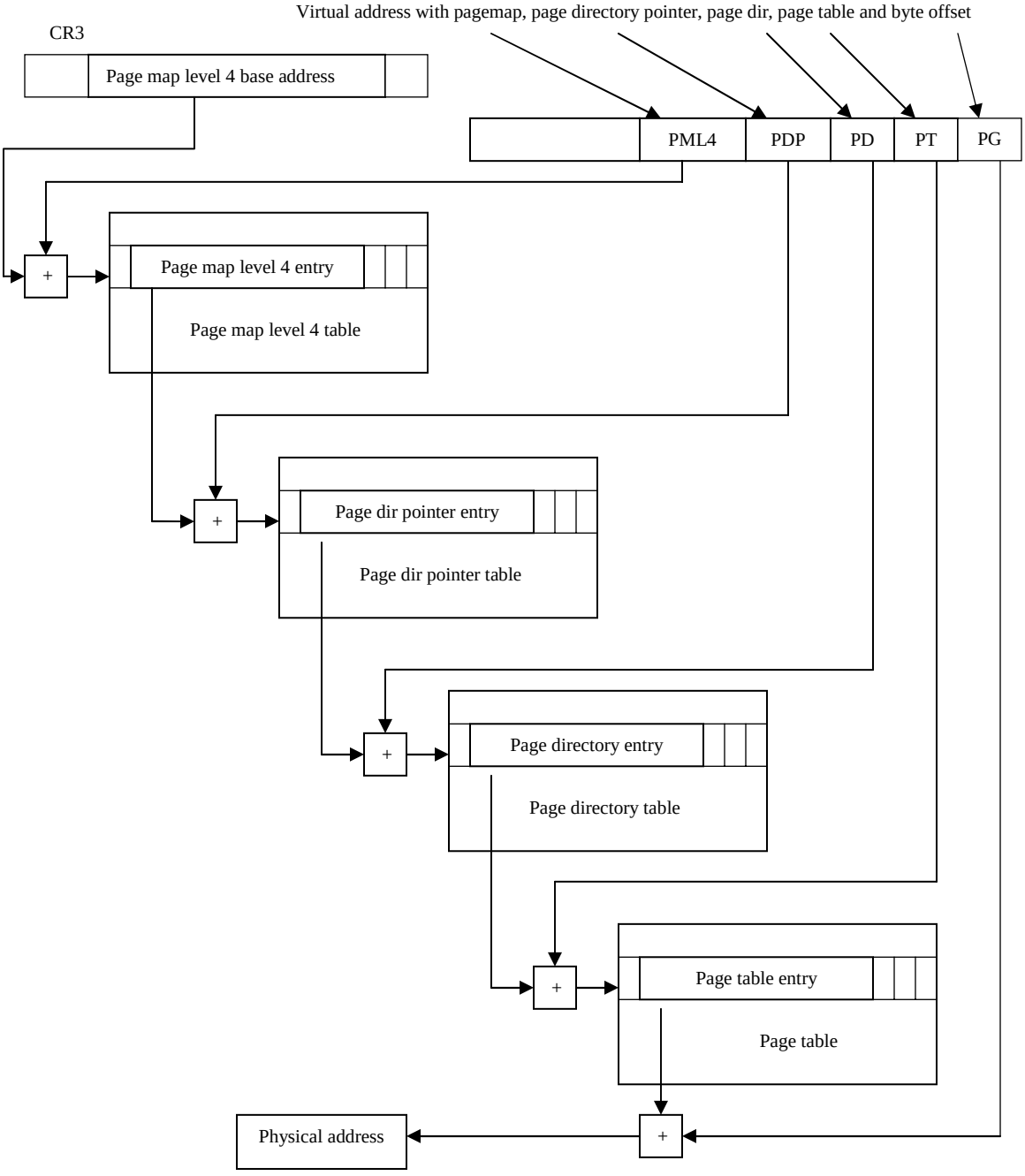

Figure 5-55. 4K-byte page translation, from linear address to physical address

When the page size is 2 M-byte, bits 0 to 20 represent the byte offset into the physical page. That is, page table offset and byte offset of the 4 K-byte page translation are combined to provide a byte offset into the 2

M-byte physical page. The following figure illustrates how paging is used to translate a 64-bit linear address into a physical address for the 2 M-byte page size.

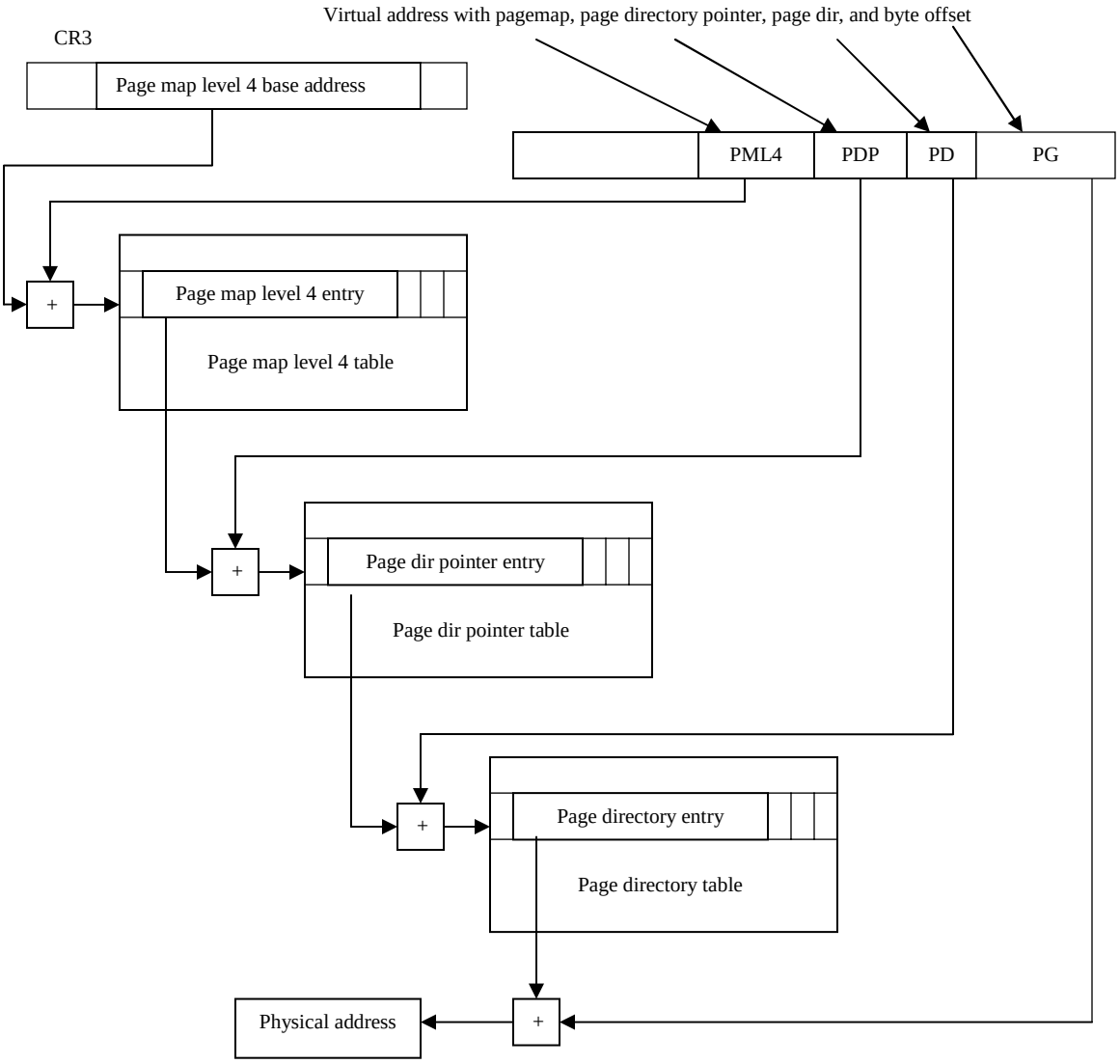

Figure 5-56. 2M-byte page translation, from linear address to physical address

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 R/W (Read/Write) flag, the U/S (User/Supervisor) flag, and the NX (No Execute) flag.

The following diagram displays 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 K-byte page and 2 M-byte page sizes.

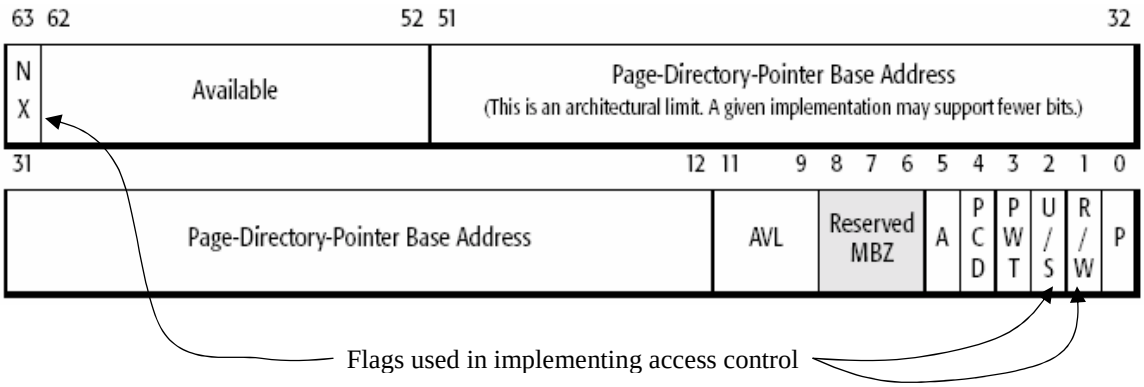

Figure 5-57. Page map level 4 entry

Read/Write flag

Read/Write flag contains the access rights of the physical pages mapped by the table entry. The Read/Write 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 Read/Write 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

User/Supervisor flag contains the privilege level that is required to access 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 it can be accessed always. 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 is set to 0, code can be executed from the mapped physical pages. Otherwise, when set to one, 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.

Effects of segmentation on page protection

SLES operates in the x86-64 architecture's compatibility mode. In compatibility mode, both segmentation and paging are used to translate a logical address into a physical address. Segement-protection and page-protection checks are performed serially. Segment-protection checks are made first and if they fail, page protection checks are not performed. Therefore, for a successful access to a physical page, both segment protection and page protection checks must succeed.

5.5.1.5.3 Translation Lookaside Buffers (TLB)

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 5.5.1.5.2, 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 5.5.1.5.2.

5.5.2 Kernel memory management

In the SLES kernel, a portion of the Random Access Memory (RAM) is permanently assigned to the 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.

In this section, we consider dynamic memory used by the kernel and highlight how the object reuse requirement is met. This section discusses the three sections of kernel memory management: Page Frame Management, Memory Area Management, and Noncontiguous Memory Area Management. For a complete description of Kernel Memory Management, please refer to the *SLES Low Level Design,* by Janak Desai, George Wilson, and Michael Halcrow.

5.5.2.1 Page frame management

The SLES kernel adopts the smaller 4 KB page-frame size 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.

5.5.2.2 Memory area management

A memory area is an arbitrary length of a sequence of memory cells that have contiguous physical addresses. Memory areas are typically smaller, few tens to hundreds of bytes, compared to page frames. Because allocating a full page frame to hold a few bytes is wasteful, the system employs a different scheme, called slab allocator, to allocate smaller memory areas. Slab allocator interfaces with the page frame allocator algorithm, Buddy System, to obtain free contiguous memory. 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.

5.5.2.3 Noncontiguous memory area management

Although it is preferable to map memory areas into sets of contiguous page frames, it makes sense to consider noncontiguous page frames accessed through contiguous linear addresses if the requests for memory areas are infrequent. Noncontiguous page frames help to reduce external fragmentation but require modification of the kernel page tables. The SLES kernel provides the `vmalloc()` function to allocate noncontiguous memory area to the kernel. To allocate memory for kernel use, `vmalloc()` calls `vmalloc_area_pages()` with a `gfp_mask` flag that is always set to `GFP_KERNEL | __GFP_HIGHMEM`.

5.5.3 Process address space

The address space of a process consists of all the linear (virtual) addresses that the process is allowed to use. The kernel allocates and maintains this address space. The address space represents contiguous groups of linear addresses through resources called memory regions, which are characterized by an initial linear address, a length, and some access rights. Although a process does not have direct control over its address space, actions taken by it can affect its address space.

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. For more detailed information, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

All information related to the process address space is included in a data structure called the memory descriptor. The memory descriptor's fields include the number of memory regions, pointers to code, data, heap, and user stack. Each memory region, represented by the `vm_area_struct` structure, identifies a linear address interval. Memory regions owned by a process never overlap. Memory regions are stored in a red-black binary tree and a simple linked list for efficient insertion and deletion of memory regions into a process's address space. As stated previously, a process's address space may need to expand or contract as a result of actions, such as the following:

- Expand heap by `malloc()` or `sbrk()` calls
- Kernel decides to increase process's user mode stack
- Creation/deletion of IPC shared memory region
- Process decides to memory map a file
- Process execs another program

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 a process's address space. 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 using the `memset()` macro, thus satisfying the object reuse requirement. The kernel also provides the `fadvise` system call, by which a process can advise the kernel on its intended access pattern for file data. This intention to read or write a certain file data allows the kernel to optimize the access. The intention conveyed to the kernel is not binding and is used for optimization only. Because access control is not performed by the system call, the system call is not part of the Trusted Security Function Interface (TSFI).

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.

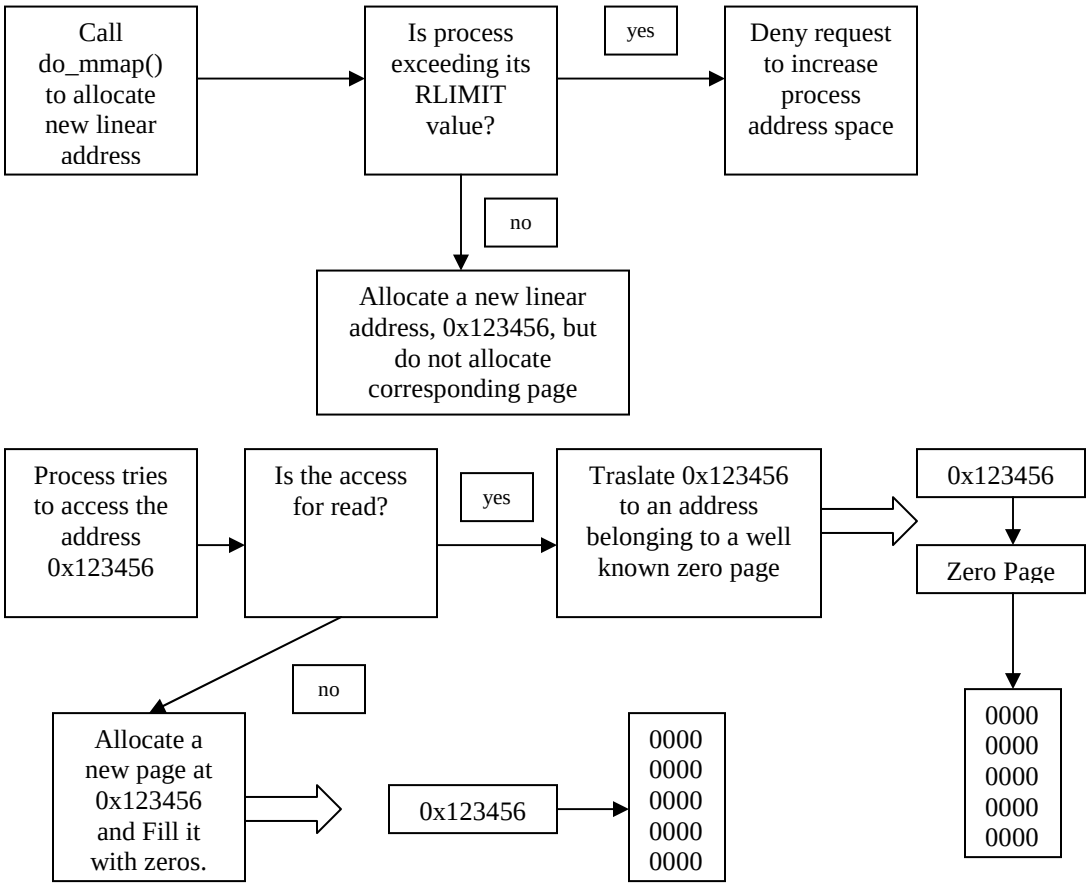

Figure 5-58. Object reuse handling while allocating new linear address

5.5.4 Symmetric multi processing and synchronization

The SLES kernel is reentrant. This means that several processes may be executing in kernel mode at the same time. Memory allocation and addressing described in the previous sections assume that access to kernel data structures by different processes is synchronized to prevent corruption. This synchronization is needed to support reentrancy and Symmetric Multi Processing (SMP – the system can use multiple processors and there is no discrimination among them). This section describes various synchronization techniques used by the SLES kernel. For more detailed information, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

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.

5.5.4.1 Atomic operations

Assembly language instructions of the type “read-modify-write” access a memory location twice, the first time to read the old value and the second time to write a new value. The SLES kernel provides a way to make such an operation atomic 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 xSeries 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.

5.5.4.2 Memory barriers

Optimization performed by compilers reorder instructions, which may affect memory access. When dealing with critical regions, instruction reordering should be avoided. A memory barrier ensures that the operations placed before the barrier, are finished before starting the operations placed after the barrier. The barrier acts like a firewall that cannot be passed by any assembly language instruction. The eServer series processors provide the following kinds of assembly language instructions that act as memory barriers:

- All instructions that operate on I/O ports.
- All instructions prefixed by the lock byte.
- All instructions that write into control registers, system registers, or debug registers.
- Memory barrier primitives, such as `mb()`, `rmb()`, and `wmb()`, that provide memory barrier, read memory barrier, and write memory barrier, respectively, for multiprocessor or uniprocessor systems.
- Memory barrier primitives, such as `smp_mb()`, `smp_rmb()`, and `smp_wmb()`, that provide memory barrier, read memory barrier, and write memory barrier, respectively, for multiprocessor systems only.

5.5.4.3 Spin locks

Spin locks are a special kind of lock designed to work in a multiprocessor environment. If the kernel control path finds the spin lock “open,” it acquires the lock and continues its execution. Conversely, if the kernel control path finds the lock “closed” by a kernel control path running on another CPU, it “spins” around, repeatedly executing a tight instruction loop, until the lock is released.

5.5.4.4 Kernel semaphores

A kernel semaphore is similar to a spin lock in that it doesn’t allow a kernel control path to proceed unless the lock is open. However, whenever a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It becomes runnable again when the resource is released.

5.6 Audit subsystem

The TOE includes a comprehensive audit subsystem, Linux Audit-Subsystem (LAuS), to provide an administrative user with the ability to identify attempted and realized violations of the system’s security policy. The audit subsystem records security relevant events in the form of an audit trail, and provides tools to an administrative user to configure the subsystem and evaluate audit records. LAuS is designed to meet the audit requirement of the Controlled Access Protection Profile (CAPP).

This section describes the operation of the audit subsystem and the high-level design of the kernel audit subsystem. For more detailed information on the low-level design of the kernel audit subsystem, please refer to *Linux Audit-Subsystem Design Documentation for Kernel 2.6*, by Thomas Biege

5.6.1 Audit subsystem operation

In order to catch all security-relevant events, the audit subsystem is designed to record actions of the SLES kernel as well as those of the security-relevant trusted 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 administrative user to irrefutably attribute the action to a particular user and ascertain the time that the action was taken. Audit records are arranged in chronological order to form an audit trail.

The audit subsystem operation consists of the following steps:

- Load the audit kernel extension module (*audit.o*).
- Launch the audit daemon (`auditd`), which reads configuration files and sets kernel audit parameters by communicating with the kernel through the audit device. The audit daemon then activates filter configuration in the kernel and in a continuous loop, reads raw data from the kernel and writes audit records to a disk log.
- Once auditing is enabled in the kernel, trusted processes that want to create audit records attach themselves to the audit subsystem. Processes such as login that perform authentication use PAM to attach themselves to the audit subsystems. Other trusted processes, such as web server or FTP server, use `aurun(8)` to attach themselves to the audit subsystem. Descendants of attached processes are automatically attached to the audit subsystem.
- Intercept potential security relevant system calls performed by processes attached to the audit subsystem. Evaluate system call event's security relevance based on audit configuration parameters. If security relevant, generates an audit record.
- Use tools such as `aucat` and `augrep` to view and analyze audit logs.

There are two sources that generate audit records: The SLES kernel and trusted user programs. The following sections describe how records are generated through these sources.

The SLES kernel

The SLES kernel evaluates each security relevant system call, file system object access, and netlink operation for the potential to generate an audit record.

The following describes a typical operation of a process that generates audit records for each security relevant system call it performs:

- To begin auditing, a process with CAP_SYS_ADMIN capability attaches itself to the audit subsystem. Once attached, every security relevant system call performed by the process is evaluated in the kernel. The process's descendents maintain their attachment to the audit subsystem. The process can only detach itself from the audit subsystem if it has the CAP_SYS_ADMIN capability.
- All security relevant system calls made by the process are intercepted at the beginning of the system call code, and at various return points to handle different errors. The intercept routine then evaluates the intended action for its security relevance. If deemed relevant, the system collects appropriate data to be used in the corresponding audit record.
- The process invokes a system call service routine to perform the intended system call.
- If the action performed is security relevant, the process invokes the audit subsystem function to generate an audit record.
- The audit record is placed in a kernel buffer; from there it is transferred to the audit trail by the audit daemon.

The following describes a typical operation of a process that generates audit records for access to file system objects:

- To begin auditing, a process with CAP_SYS_ADMIN capability attaches itself to the audit subsystem. Once attached, every time the process accesses a file system object through VFS

functions such as open, truncate, and chdir, audit intercept functions are called at the beginning and at the end of the monitored VFS functions.

- The process completes the intended VFS operation.
- If the operation performed was security relevant, the process invokes the audit subsystem function to generate an audit record.
- The audit record is placed in a kernel buffer; from there it is transferred to the audit trail by the audit daemon.

The following describes a typical netlink operation that generates an audit record:

- The kernel encounters a netlink message.
- If the netlink message is routing related, it invokes an audit subsystem function to collect appropriate data to be used for the audit record.
- Once the netlink message is processed, it invokes an audit subsystem function to generate an audit record.
- The audit record is placed in a kernel buffer; from there it is transferred to the audit trail by the audit daemon.

The following diagram schematically describes the flow of data that results in audit records generated by the kernel.

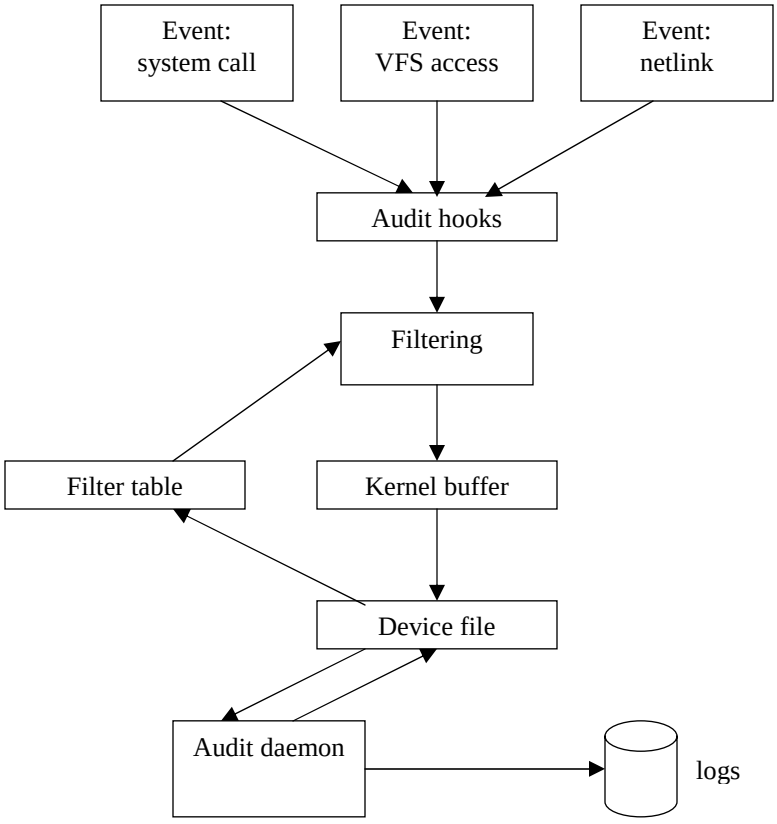

Figure 5-59. Audit data flow in the kernel

Trusted programs

Trusted programs, such as those that perform authentication, create their own audit records that describe their actions. Because trusted programs are trusted, the kernel does not need to audit each and every system call they perform. The following describes a typical trusted program operation with respect to audit:

- To begin auditing, the process associated with a trusted program attaches itself to the audit subsystem. The process's descendants maintain their attachment to the audit subsystem. The process can only detach itself from the audit subsystem if it has the CAP_SYS_ADMIN capability.
- Once attached, the process suspends system call auditing to prevent the kernel from generating audit records for each security relevant system call performed by the process.
- After performing security-relevant actions, the program formats the audit record describing the action and sends it to the audit trail with the help of the kernel.

The following diagram schematically describes the data flow that results in audit records generated by trusted programs.

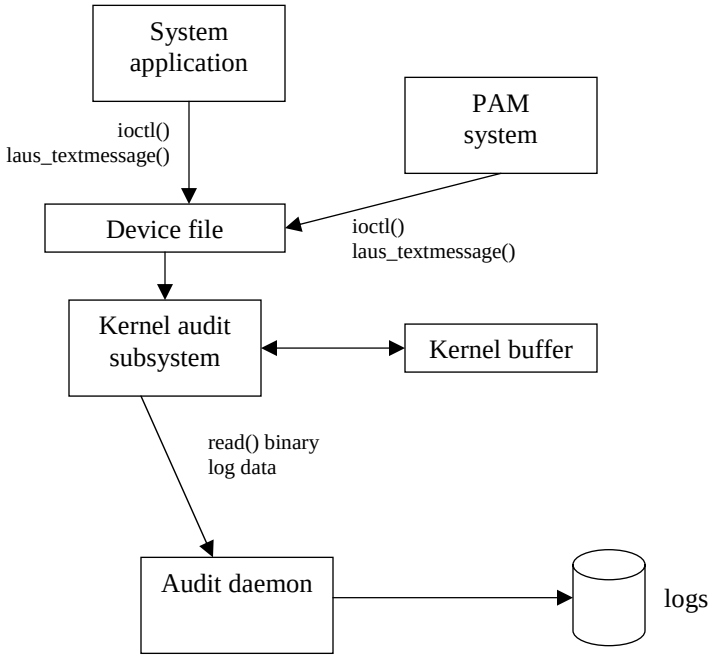

Figure 5-60. Audit data flow from the user space

Components of the audit subsystem
The following illustrates different components that make up the audit subsystem and how they interact with each other to implement functionality required by the Controlled Access Protection Profile.

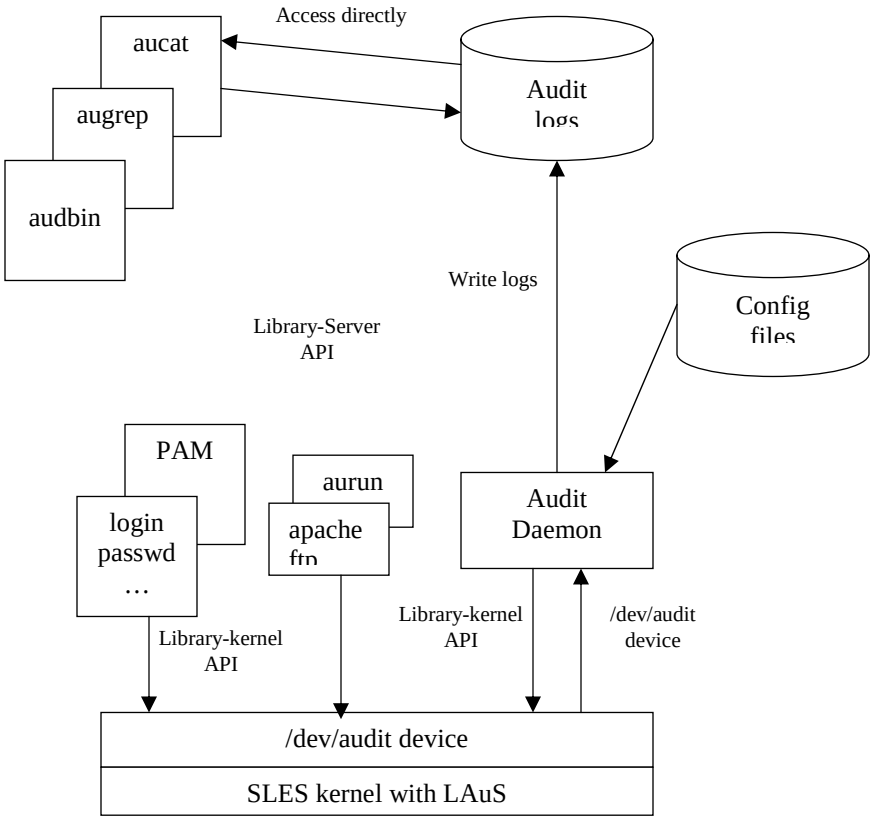

Figure 5-61. Linux Audit-Subsystem (LAuS) components

These components can be placed into three different subsystems. The SLES kernel with LAuS is part of the kernel audit subsystem and is described below in section 5.6.2. The audit device driver is part of the device driver subsystem of the kernel and is described in section 5.8.3. The rest form the user level audit subsystem, which is described in section 5.14.

5.6.2 SLES kernel with LAuS

The kernel component of LAuS consists of extensions to process task structures for storing additional audit related attributes, two intercept functions, which are placed at the beginning and at various exit points of security relevant system calls and object access functions of VFS, an extension to routing changes, and the addition of an audit device driver.

Task structure extensions
Each process is represented by a task structure in the kernel. This task structure is extended to add a pointer to audit data for that process. The audit data is represented by the structure `aud_process`, which contains the following 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 their actions, even across su(8) calls or use of setuid binaries.

Audit ID
Audit ID is a unique session identifier that is assigned to every process attached to the audit subsystem. If a process attached to the audit subsystem forks a child, the child process inherits the audit ID of the parent. Audit ID is used by an administrative user to group together actions performed by one session.

suspended flag
The `suspended` flag is used to indicate if the process has suspended system call auditing.

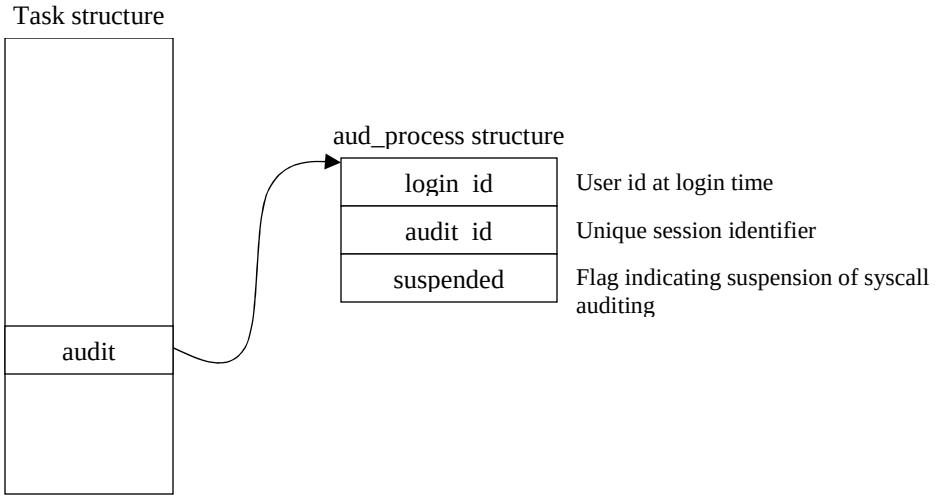

Figure 5-62. LAuS extensions to task structure

Audit intercept functions

In order to evaluate potential security relevant system calls as audit candidates, a subset (table 5-1) of the SLES kernel's system calls are modified with calls to audit intercept routines. The following table lists system calls whose service routines are modified to call audit intercept functions.

Syscall call Name
access
adjtimex
brk
capset
chdir
chmod
chown
clone
creat
delete_module
execve
fchmod
fchown
fork
fremovexattr
fsetxattr
init module
ioctl
ioperm
iopl

Syscall call Name
ipc (msgctl, msgget, semctl, semget, shmat, shmctl, shmget)
lchown
link
lremovexattr
lsetxattr
mkdir
mknod
mount
open
ptrace
removexattr
rename
rmdir
semtimedop
setfsgid
setfsuid
setgid
setgroups
setregid
setresgid
setresuid
setreuid
settimeofday
setuid
setxattr
socketcall (bind)
swapon
symlink
truncate
umask
unlink
utime/utimes
vfork

Table 5-1. System calls modified audit intercept functions

Audit intercept functions are inserted at the beginning and at appropriate exit points of the system call service function. 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 function to perform the system call. The third step switches from the kernel to user space after copying the result of the system call. The LAuS extensions to the previous steps involve calling the audit subsystem function `audit_intercept()` at the beginning of step two, and calling the audit subsystem function `audit_result()` at various exit points of steps two. `audit_intercept()` stores relevant audit data needed to create the audit record. `audit_result()`, which is invoked after the system call action is performed, applies filtering logic, generates the audit record, and places it in the kernel buffer.

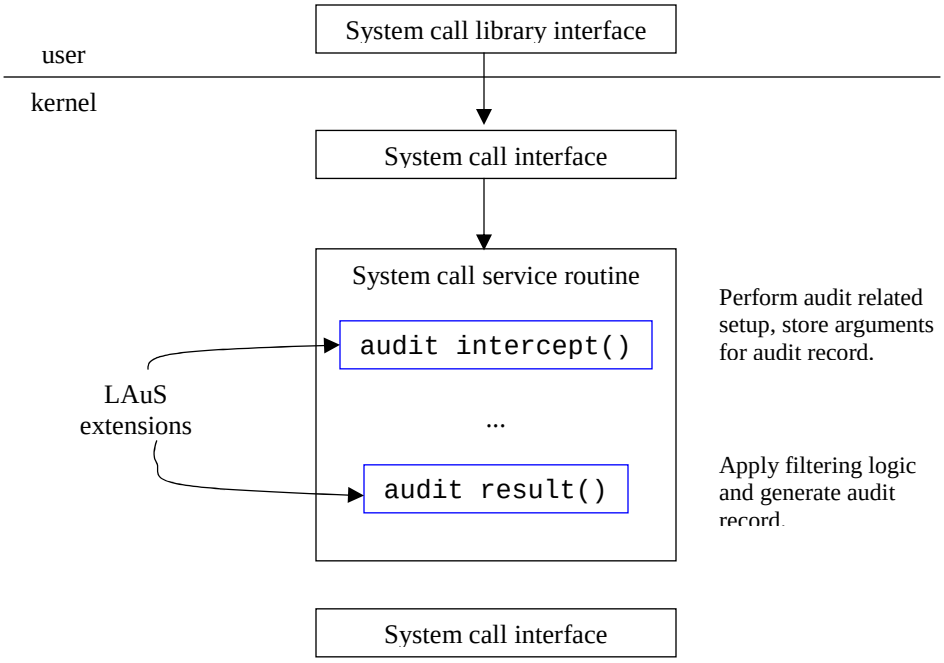

Figure 5-63. Extensions to potential security relevant system call interface

Filtering logic allows an administrative user to filter out events based on user ID, system call, and file names, using predicates and logical operations. Basic predicates can be combined to create more complex user defined predicates. For example:

```
predicate is-one-or-two = eq(1) || eq(2);
```

The predicates can be used by defining a filter or by attaching the predicate to a syscall.

```
filter uid-is-one-or-two = is-one-or-two(uid);
...
syscall sleep = is-one-or-two(arg0);
```

The filter is used to bind the predicate to a so called target (syscall argument, process property, syscall result, etc.). In order to handle a class of objects more easily, the audit filter lets you specify a *set*.

```
set sensitive = { /etc, /root, /usr }
...
predicate is-sensitive = prefix(@sensitive);
```

The example above illustrates the use of sets. A set can be referenced by a leading `@` sign. Please refer to man page `audit-filter.conf(5)` for a more detailed description of the filtering scheme.

Extensions to routing changes

The SLES kernel supports two mechanisms for configuring IP network devices and IP routing. The first mechanism, using the `ioctl()`system call, does not need any specific audit related changes because the `ioctl()`system call is audited as part of the system call audit. The second method, using `AF_NETLINK` sockets, is extended for audit to record routing changes. Netlink messages are sent through sockets of type AF_NETLINK, where the destination is identified by numeric IDs such as `NETLINK_ROUTE`. The audit code taps into the function `rtenetlink_rcv_skb()`, which delivers `NETLINK_ROUTE` messages.

The audit function is invoked after the netlink message is processed. The message length, message outcome, and the message itself are passed on to the audit subsystem for inspection. Based on the message and its outcome, the audit subsystem decides if an audit record is to be generated.

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 features at the kernel's upper layer. Lower-layer function, 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:

- The kernel notices that a requested feature is not resident in the kernel.
- The kernel executes the *modprobe* program to load a module that fits this symbolic description.
- *modprobe* looks into its internal "alias" translation table to see if there is match. This table is configured by "alias" lines in */etc/modprobe.conf*.
- *modprobe* then inserts the modules that the kernel needs. The modules are configured according to options specified in the */etc/modprobe.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 provides 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.

The */etc/modprobe.conf* file can only be modified by an authorized user, allowing the administrator complete control over which modules can be loaded and with what configuration options. In the kernel, the module load function `sys_init_module` (called by *modprobe* during automatic load operation) is protected by a capability check of `CAP_SYS_MODULE`. Thus, only a privileged process can initiate the loading of modules in the kernel. In order to maintain backward compatibility, the 2.6 kernel stores enhanced 2.6 versions of module loading tools in the */etc/modprobe.d*, while providing /etc/modutils for the 2.4 versions. The format of the */etc/modprobe.conf* allows inclusion of other files with the "`include` *filename*" command. This allows administrators to split modules into separate configuration files. Administrators can keep all critical modules in */etc/modprobe.conf* and keep local customization in */etc/modprobe.conf.local*.

Loadable security modules (LSM)

The Linux kernel, from version 2.6, provides a flexible infrastructure for implementing access control policies. This infrastructure takes the form of a set of security mediations strategically located throughout the kernel. These generic security mediations allow kernel modules to implement additional restrictive access control policies. To avoid performance penalties in the absence loadable security modules, security mediations are compiled in to the kernel as inline functions that invoke LSM functions through a global security operations structure of type `security_operations`. A loadable security module registers itself with the kernel by providing its own `security_operations` structure. The `security_operations` structure supplied during module registration contains valid function pointers to those security mediation functions that the module wants to define. The process is similar to different file systems registering themselves with the VFS by providing their own `file_system_type` structure.

The security mediations are of two forms. The first type is called after a kernel object, such as a file or a process, is created or modified. The security mediation allows a security module to create or update any security associations for that object. The second type is called before access to kernel objects in order to allow a security module to deny that access.

The following illustrates how a loadable security module, called xyz, can implement additional access control policy.

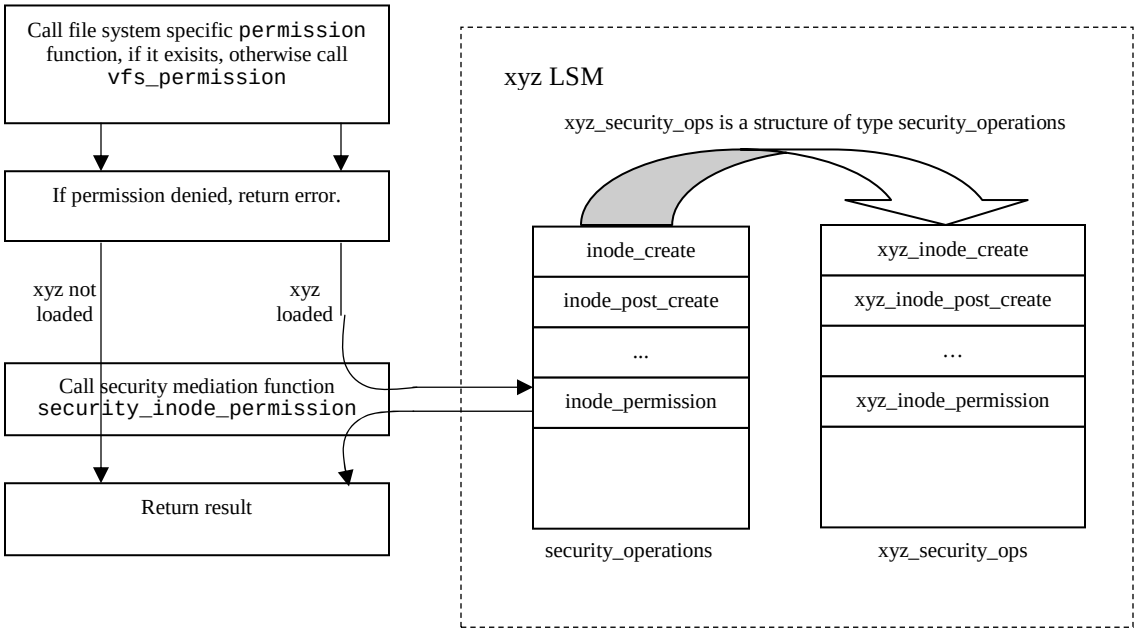

Figure 5-64. Loadable Security Modules

From version 2.6, Linux process capabilities are implemented using an LSM module. The common capability module gets loaded on top of the kernel during initialization and contains functions to manage and enforce process capabilities.

5.8 Device drivers

A device driver consists of data structures and functions that make a hardware device respond to a well-defined programming interface. The kernel interacts with the device only through this well-defined interface. It allows the kernel to control the device without knowing the underlying device specific functions. For detailed information on device drivers, please refer to the *SLES Low Level Design,* by Janak Desai, George Wilson, and Michael Halcrow.

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 and I/O and the Networking subsystem, have direct access to these device driver functions because the device driver is loaded into the kernel space at system initialization time. The File and I/O subsystem and the Networking subsystem 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 File and I/O subsystem buffer cache. 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 iSeries and zSeries supports virtual devices. From the perspective of the SLES kernel, these devices are treated no differently than other devices. That is, the SLES kernel thinks that it is controlling devices directly. However, the hypervisor on iSeries and the z/VM on zSeries map these virtual devices to real devices, allowing SLES access to devices supported by OS/400 when running on iSeries, and devices supported by z/VM when running on zSeries. The following subsections briefly describe this virtualization of I/O, followed by brief description of device drivers for audit device, character device, and block device.

5.8.1 I/O virtualization on iSeries

SLES runs on iSeries in a logical partition. In a logical partition, devices can operate in two different modes, native I/O and virtual I/O. In native I/O mode, SLES device driver directly control the device as described in the section above. In virtual I/O mode, SLES uses generic device drivers to communicate with real devices. For example, a generic disk driver is used to communicate with different types of disks. This virtualization of I/O is implemented by the iSeries hypervisor utilizing the special I/O structure of the iSeries hardware.

On iSeries, many I/O events are not delivered as interrupts but rather as “events” on a queue. The hypervisor provides a mechanism for communicating between logical partitions using these events. These logical partition events (LP events) are anchored off processor architecture control area. The LP events are key to the I/O structure of Linux on iSeries. The logical partition running SLES, first identifies the hosting partition by making a hypervisor call, and then send all virtual I/O events to that partition.

5.8.2 I/O virtualization on zSeries

SLES runs on zSeries 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. The virtual machine simulates the existence of a dedicated real machine including storage and I/O resources. Virtual machines can run applications or even operating systems. SLES on z/Series runs in such a virtual machine provided by the z/VM operating system. z/VM provides hypervisor functions through its Control Program (CP). The Control Program prevents guest virtual machines from interfering with each other. This isolation is implemented using the interpretive execution facility of the zSeries hardware. The Processor Resource/System Manager (PR/SM) permits a virtual machine instruction stream to be run on the processor using a single instruction, SIE. The SIE instruction is used by the machine’s logical partitioning support functions to divide a zSeries processor complex into logical partitions. When the Control Program dispatches a virtual machine, details about the virtual machines are provided to the hardware. The SIE instruction runs the virtual machine until the virtual machine’s time slice has been consumed, or the virtual machine wants to perform an operation for which the Control Program must regain control. In this way, the full capabilities and speed of the CPU are available to the virtual machine and only those instructions that require assistance from or validation by the Control Program are intercepted.

To virtualize devices, the Control Program acts as a barrier between virtual machines and devices to which the Control Program has access. The Control Program mediates access to those real devices based on configuration of the device as shared, or exclusive use for a particular virtual machine. When a virtual machine makes an I/O request, the request is intercepted by the Control Program, such that the virtual

addresses in the I/O request can be translated to their corresponding real memory addresses. The Control Program validates the request for access control and then starts the I/O operation on behalf of the virtual machine.

For additional detail on z/VM features and technical details, please refer to the following:

Alan Altmark and Cliff Laking. The value of z/VM: Security and Integrity
http://www.ibm.com/servers/eserver/zseries/library/techpapers/pdf/gm130145.pdf

z/VM general Information
http://www.vm.ibm.com/pubs/pdfs/HCSF8A60.PDF

5.8.3 Audit device driver

To enable bidirectional communications between user space and kernel space, LAuS provides an audit device driver. User space programs access this device driver through the */dev/audit* device-special file. The device has the major number 10 and minor number 224.

5.8.4 Character device driver

A character device driver is a collection of routines that make a character device, such as a mouse or a keyboard, respond to a well-defined programming interface.

Programs operate on character devices by opening their file system entry. The 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 `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 `f_op` field of the file object points to the appropriate `file_operations` defined for that particular character device. The structure file_operations provides pointers to generic operations that can be performed on a file. Each device driver defines these file operations that are valid for the type of device that the driver manages.

This extra level of indirection is needed for character devices and not block devices because of the large variety of character devices and the operations that 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.

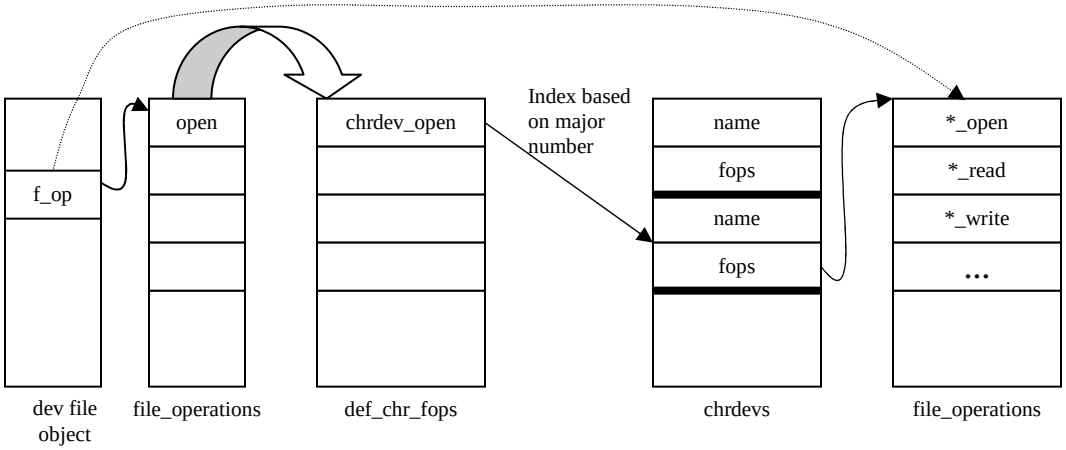

Figure 5-65. Setup of `f_op` *for character device specific file operations*

5.8.5 Block device driver

A block device driver is a collection of routines that make a block device, such as a disk drive, respond to a well-defined programming interface.

Programs operate on block devices by opening their file system entry. The file system entry contains a major and a minor number by which the kernel identifies the device. The kernel maintains a hash table, indexable 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 a structure `block_device_operations`. Structure block_device_operations contains methods to `open`, `release`, `llseek`, `read`, `write`, `mmap`, `fsync`, and `ioctl` the block device. Each block device driver needs to implement these block device operations for 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.

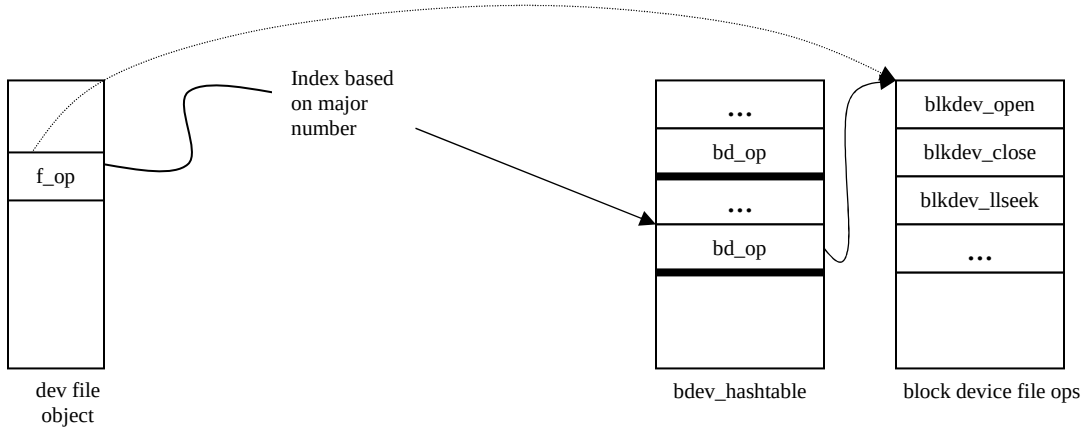

Figure 5-66. Setup of `f_op` *for block device specific file operations*

5.9 System initialization

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 the xSeries, pSeries, iSeries, zSeries, and eServer 325 lines of servers, which are all part of the TOE.

5.9.1 xSeries

This section briefly describes the system initialization process for xSeries servers. For detailed information on system initialization, please refer to the following:

Booting Linux: History and the Future, 2000 Ottawa Linux Symposium, by Almesberger, Werner
SLES Low Level Design, by Janak Desai, George Wilson, and Michael Halcrow
/usr/src/linux/Documentation/i386/boot.txt

5.9.1.1 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.

5.9.1.2 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. A typical boot image on various UNIX variant systems consists of a kernel binary that is combined with an initial file system image. SLES does not need to combine a kernel binary with the initial file system because it supports the boot loader GRUB (GRand Unified Boot Loader). GRUB lets you set up pointers in the boot sector to the kernel image and to the RAM file system image. GRUB is considered to be a part of the TSF. For detailed information on GRUB, please refer to the following:

http://www.gnu.org/manual/grub-0.92/html_mono/grub.html
/usr/share/info/grub.info

5.9.1.3 Boot process

For an individual computer, 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 the TOE.
2. Initializes hardware devices and makes sure they operate without IRQ or I/O port conflicts.
3. Searches for the operating system to boot in an order predefined by the BIOS setting. Once a valid device is found, 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. Reprograms the Programmable Interrupt Controller and maps the 16 hardware interrupts to the range of vectors from 32 to 47. Switches the CPU from Real Mode to Protected Mode and then jumps to the `startup_32()` function.
7. Initializes the segmentation registers and provisional stack. Fills the area of uninitialized data of the kernel with zeros.
8. Decompresses the kernel, moves it into its final position at 0x00100000, and jumps to that address.
9. Calls the second `startup_32()` function to set up the execution environment for process 0.
10. Initializes the segmentation registers.
11. Sets up the kernel mode stack for process 0.
12. Initializes the provisional Page Tables and enables paging.
13. Fills the bss segment of the kernel with zeros.
14. 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.
15. 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 `start_kernel()` function.
16. `start_kernel()` completes the kernel initialization by initializing Page Tables, Memory Handling Data Structures, IDT tables, slab allocator (described in section 5.2.2.2), system date, and system time.

17. Uncompresses the initial RAM file system *initrd*, mounts it, and then executes */linuxrc*.
18. Unmounts *initrd*, mounts the root file system, and executes */sbin/init*. 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*.

    - Allows an administrator to perform interactive debugging of the startup process by executing the script */etc/sysconfig/boot.*
    - Mounts special file system */proc*.
    - Mounts special file system */dev/pts*.
    - Executes */etc/init.d/boot.local* set up by an administrator to perform site-specific setup functions.
    - If file */var/lib/YaST2/runme_at_boot* exists, finishes YaST2 installation from previous session.
    - If file */var/lib/YaST2/run_suseconfig* exists, executes */sbin/SuSEconfig* to configure the operating system.

20. Performs run-level specific initialization by executing startup scripts from */etc/init.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. For more details on services started at run level 3, please refer to the scripts in */etc/init.d/rc3.d*.

    - Saves and restores the system entropy tool for higher quality random number generation.
    - Starts the `atd` daemon.
    - Configures network interfaces.
    - Starts the system logging daemons.
    - Starts the `sshd` daemon.
    - Starts the `cron` daemon.
    - Probes hardware for setup and configuration.
    - Starts the program `agetty.`

The following diagram schematically describes the boot process.

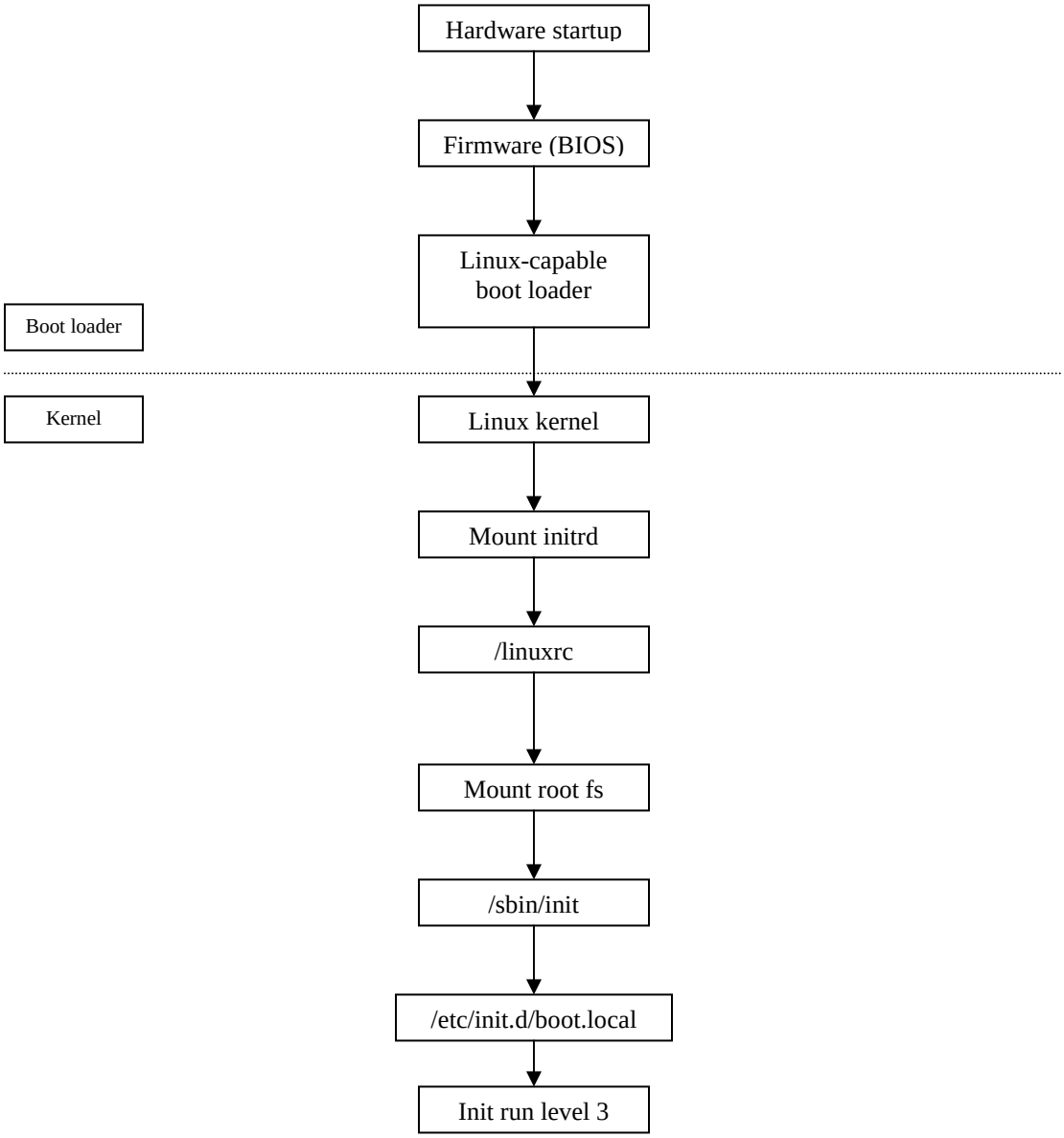

Figure 5-67. xSeries SLES boot sequence

5.9.2 pSeries

This section briefly describes the system initialization process for pSeries servers.

5.9.2.1 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. The level of detail in the boot process described here does not include differences between the boot process on an unpartitioned system and the boot process in a logical partition. Those differences are minor and explained in the low-level design.

5.9.2.2 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. pSeries systems boot using a boot loader called Yaboot (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, please refer to the following:

http://penguinppc.org/projects/yaboot

5.9.2.3 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. 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. Interacts with OpenFirmware and determines the system configuration, including real memory layout and the device tree.
4. 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. Relocates the kernel to real address 0x0.
6. Creates the initial kernel stack and initializes TOC and `naca` pointers.
7. Builds the hardware page table (HPT) and the segment page table (STAB) to map real memory from 0x0 to HPT itself.
8. Enables relocation.
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 5.2.2.2), system date, and system time.
11. Uncompresses the initial RAM file system *initrd*, mounts it, and then executes */linuxrc*.
12. Unmounts *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/init.d/boot*.

    - Allows an administrator to perform interactive debugging of the startup process by executing the script */etc/sysconfig/boot.*
    - Mounts special file system */proc*.
    - Mounts special file system */dev/pts*.
    - Executes */etc/init.d/boot.local* set up by an administrator to perform site-specific setup functions.
    - If file */var/lib/YaST2/runme_at_boot* exists, finishes YaST2 installation from a previous session.
    - If file */var/lib/YaST2/run_suseconfig* exists, executes */sbin/SuSEconfig* to configure the operating system.

14. Performs run-level specific initialization by executing startup scripts from */etc/init.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. For more details on services started at run level 3, please refer to the scripts in */etc/init.d/rc3.d.*

    - Saves and restores the system entropy tool for higher quality random number generation.
    - Starts the `atd` daemon.

- Configures network interfaces.
- Starts the system logging daemons.
- Starts the `sshd` daemon.
- Starts the `cron` daemon.
- Probes hardware for setup and configuration.
- Starts the program `agetty.`

The following diagram schematically describes the boot process.

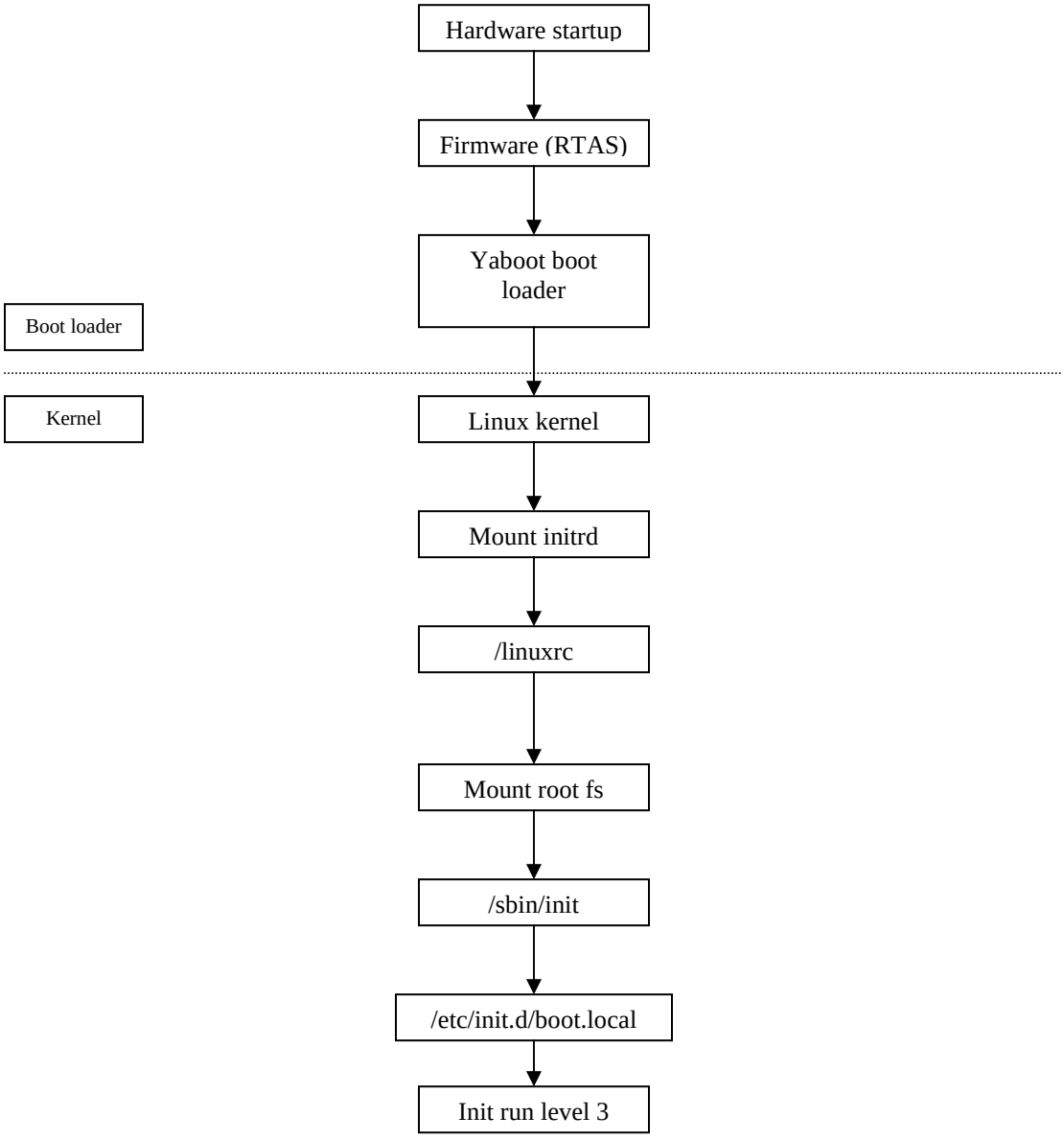

Figure 5-68. pSeries SLES boot sequence

5.9.3 iSeries

This section briefly describes the system initialization process for iSeries servers. For detailed information on iSeries LPAR initialization, please refer to the following IBM Redbook:

LPAR Configuration and Management – Working with IBM eServer iSeries Logical Partitions
http://www.redbooks.ibm.com/redbooks/pdfs/sg246251.pdf

5.9.3.1 Boot methods

On an IBM eServer, iSeries SLES runs in a secondary logical partition with primary partition running OS/400 or i5/OS, and providing hypervisor functions. Two environments are possible for running SLES in a secondary LPAR, *hosted* and *non-hosted*.

Hosted environment
In a *hosted environment*, SLES depends on OS/400 partition for some or all of its I/O. The partition is booted up and shutdown from OS/400 using *Network Server Descriptor* (NWSD). NWSD, which resides on the hosting partition, contains configuration for starting and stopping SLES partitions, and provides a link between SLES and its virtual disks. SLES is booted by *varying-on* (activating) its corresponding NWSD.

Non-hosted environment
In non-hosted environment all I/O is native and controlled by SLES device drivers. NWSD is used only to perform installation on a local disk drive. Once installation is complete, SLES boots from the partition's Direct Attached Storage Device (DASD). OS/400 running in primary partition is only used to provide low-level hypervisor functions. The partition running SLES is initialized using "Work with Partition" utility running on the primary partition.

5.9.3.2 Hypervisor

SLES runs in a logical partition on an iSeries system. Logical partitions are created by the hypervisor program that interacts with actual hardware, and provides virtual versions of hardware to operating systems running in different logical partitions. As part of an IPL (Initial Program Load), the hypervisor performs certain initializations, listed below in section 5.9.3.3, before handing control over to the operating system.

5.9.3.3 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. The hypervisor assigns memory to the partition as a 64 MB contiguous load area and the balance in 256 KB chunks.
2. 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's 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 5.2.2.2), system date, and system time.
11. Uncompresses the initial RAM file system *initrd*, 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/init.d/boot*:

    - Allows an administrator to perform interactive debugging of the startup process by executing the script */etc/sysconfig/boot*.
    - Mounts special file system */proc*.
    - Mounts special file system */dev/pts*.
    - Executes */etc/init.d/boot.local* set up by an administrator to perform site-specific setup functions.
    - If file */var/lib/YaST2/runme*_at_boot exist, finished YaST2 installation from previous session.
    - If file */var/lib/YaST2/run_suseconfig* exist, executes */sbin/SuSEconfig* to configure the operating system.

14. Performs run-level specific initialization by executing startup scripts from */etc/init.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. For more details on services started at run level 3, please refer to the scripts in */etc/init.d/rc3.d*.

    - Saves and restores the system entropy tool for higher quality random number generation.
    - Starts the `atd` daemon.
    - Configures network interfaces.
    - Starts the system logging daemons.
    - Starts the `sshd` daemon.
    - Starts the `cron` daemon.
    - Probes hardware for setup and configuration.
    - Starts the program `agetty`.

The following diagram schematically describes the boot process.

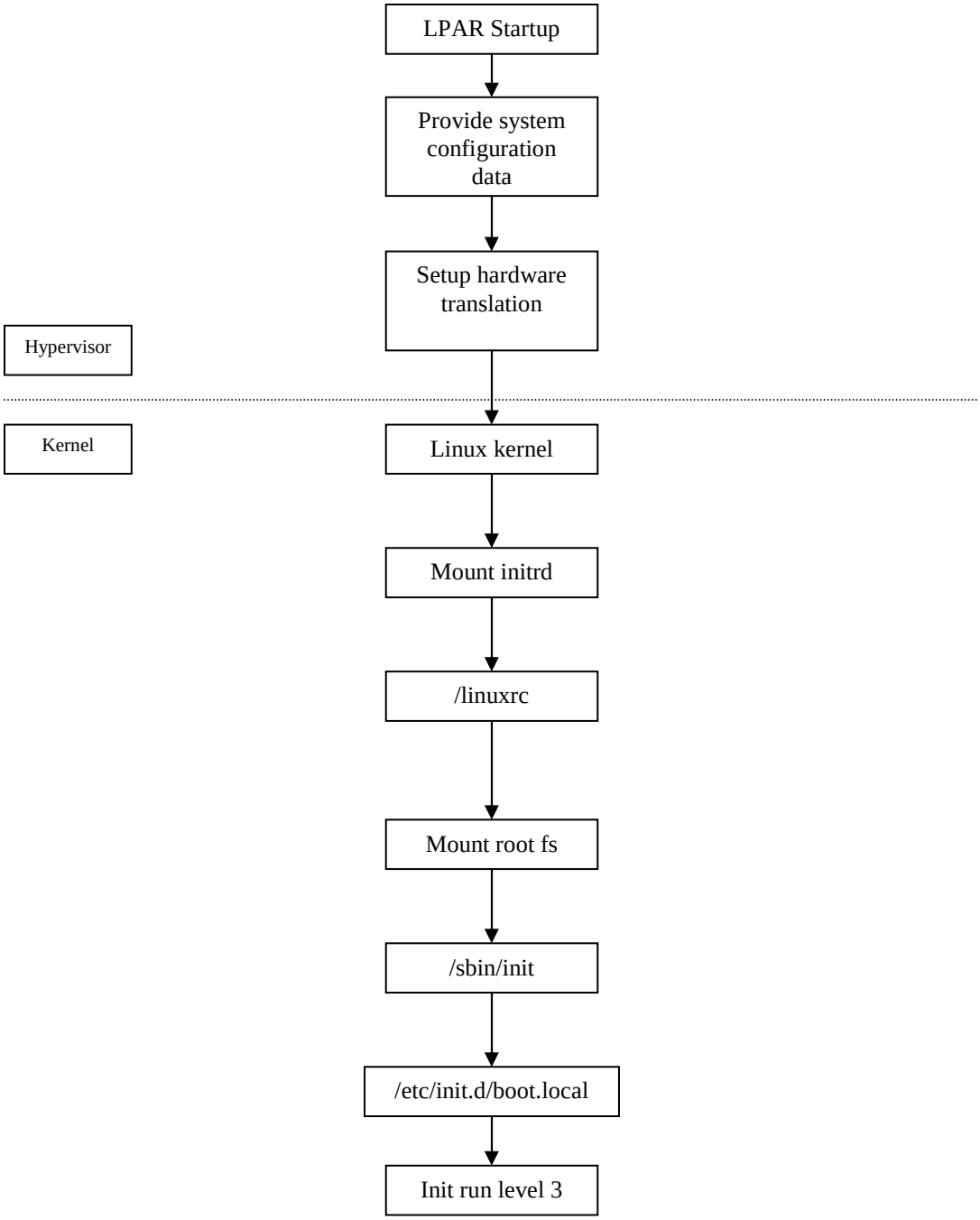

Figure 5-69. iSeries SLES boot sequence

5.9.4 zSeries

This section briefly describes the system initialization process for zSeries servers. For detailed information on the zSeries initialization, please refer to the following IBM redbook:

Linux on IBM eServer zSeries and S/390: Building SUSE SLES8 Systems under z/VM
http://www.redbooks.ibm.com/redpapers/pdfs/redp3687.pdf

5.9.4.1 Boot methods

Linux on zSeries supports three installation methods, native installation, LPAR installation or z/VM guest installation. 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(8) Linux utility is responsible for creating the boot record used during the IPL process.

5.9.4.2 Control program

On a zSeries system, SLES runs in as a guest of the z/VM operating system. The control program, which is the zSeries 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.

5.9.4.3 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(8) 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. SELS auto detects all the devices attached to the system.
4. Obtains information about the cpu(s).
5. Obtains information about disk devices and disk geometry.
6. Obtains information about network devices.
7. Jumps to `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 5.2.2.2), system date, and system time.
9. Uncompresses the initial RAM file system *initrd*, mounts it, and then executes */linuxrc*.
10. Unmounts *initrd*, mounts the root file system, and executes */sbin/init*. 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/init.d/boot*.

    - Allows an administrator to perform interactive debugging of the startup process by executing the script */etc/sysconfig/boot.*
    - Mounts special file system */proc.*
    - Mounts special file system */dev/pts*.
    - Executes */etc/init.d/boot.local* set up by an administrator to perform site-specific setup functions.
    - If file */var/lib/YaST2/runme_at_boot* exists, finishes YaST2 installation from a previous session.
    - If file */var/lib/YaST2/run_suseconfig* exists, executes */sbin/SuSEconfig* to configure the operating system.

12. Performs run-level specific initialization by executing startup scripts from */etc/init.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. For more details on services started at run level 3, please refer to the scripts in */etc/init.d/rc3.d*.

    - Saves and restores the system entropy tool for higher quality random number generation.
    - Starts the `atd` daemon.
    - Configures network interfaces.
    - Starts the system logging daemons.
    - Starts the `sshd` daemon.
    - Starts the `cron` daemon.
    - Probes hardware for setup and configuration.
    - Starts the program `agetty.`

The following schematically describes the boot process for SLES as a z/VM guest.

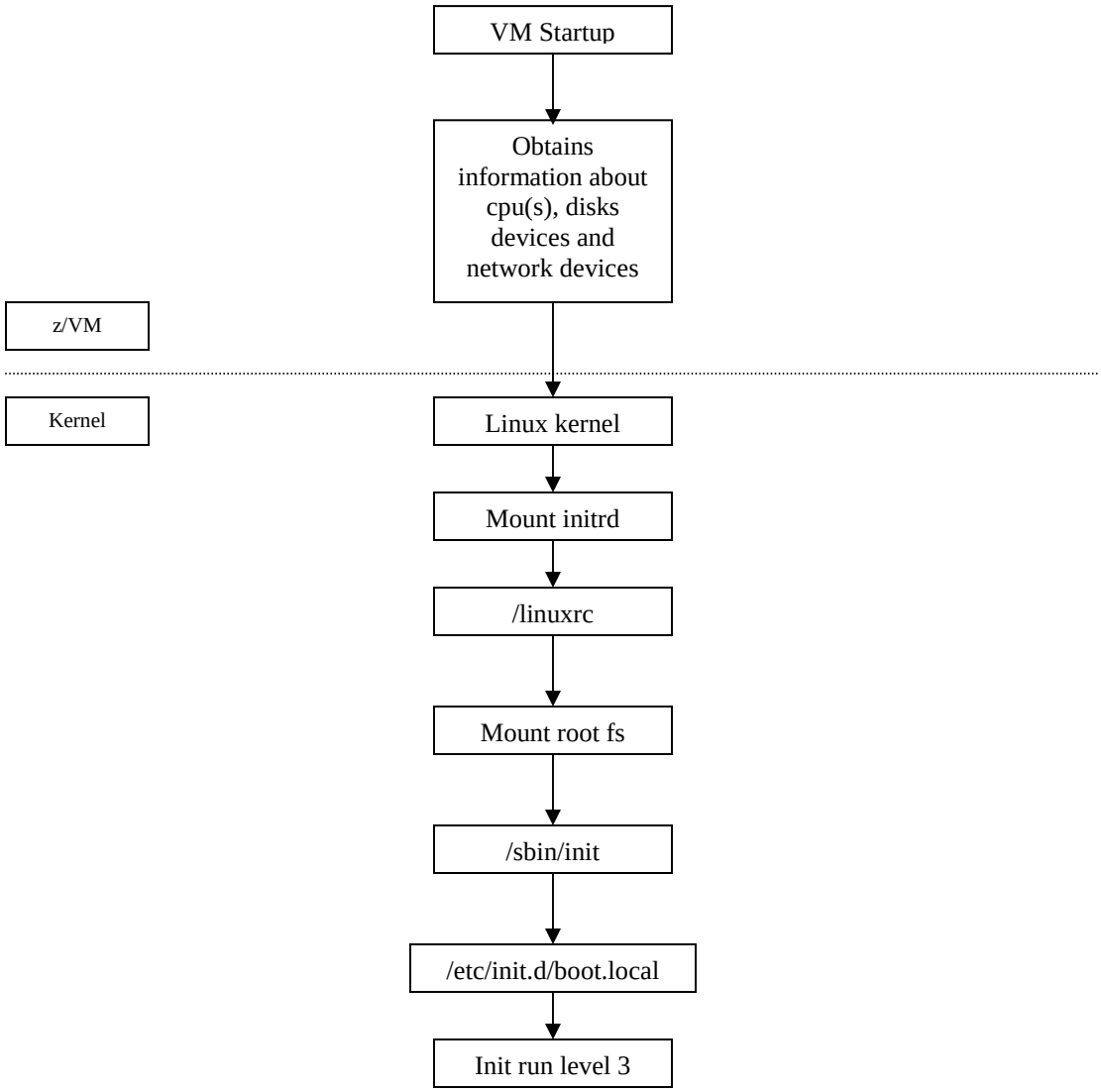

Figure 5-70. pSeries SLES boot sequence

5.9.5 eServer 325

This section briefly describes the system initialization process for eServer 325 servers. For detailed information on system initialization, please refer to the following:

AMD64 Architecture, Programmer's Manual Volume 2: System Programming,
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

5.9.5.1 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.

5.9.5.2 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. A typical boot image on various UNIX variant systems consists of a kernel binary that is combined with an initial file system image. SLES does not need to combine a kernel binary with the initial file system because it supports the boot loader GRUB (GRand Unified Boot Loader). GRUB lets you set up pointers in the boot sector to the kernel image and to the RAM file system image. GRUB is considered to be a part of the TSF. For detailed information on GRUB, please refer to the following:

http://www.gnu.org/manual/grub-0.92/html_mono/grub.html
/usr/share/info/grub.info

5.9.5.3 Boot process

For an individual computer, the boot process consists of the following steps when the CPU is powered on or reset:

1. BIOS probes hardware, establishes which devices are present, and runs Power-On Self Test (POST). BIOS is not part of the TOE.
2. Initializes hardware devices and makes sure they operate without IRQ or I/O port conflicts.
3. Searches for the operating system to boot in an order predefined by the BIOS setting. Once a valid device is found, 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. Reprograms the Programmable Interrupt Controller and maps the 16 hardware interrupts to the range of vectors from 32 to 47. Switches the CPU from Real Mode to Protected Mode and then jumps to the `startup_32()` function.
7. Initializes the segmentation registers and provisional stack. Fills the area of uninitialized data of the kernel with zeros.
8. Decompresses the kernel, moves it into its final position at 0x00100000, and jumps to that address.
9. Calls the second `startup_32()` function to set up the execution environment for process 0.
10. Prepares to enable long mode by enabling Physical Address Extensions (PAE) and Page Global Enable (PGE).
11. Sets up early boot stage 4 level page tables, enables paging, and Opteron long mode. 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 Global Descriptor Table Register (GDTR) and Local Descriptor Table Register (LDTR) 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 sections 5.5.1.5 and 5.5.2), system date, and system time.
18. Uncompresses the initial RAM file system *initrd*, 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/init.d/boot*.

    - Allows an administrator to perform interactive debugging of the startup process by executing the script */etc/sysconfig/boot.*
    - Mounts special file system */proc.*
    - Mounts special file system */dev/pts*.
    - Executes */etc/init.d/boot.local* set up by an administrator to perform site-specific setup functions.
    - If file */var/lib/YaST2/runme_at_boot* exist, finishes YaST2 installation from previous session.
    - If file */var/lib/YaST2/run_suseconfig* exist, executes */sbin/SuSEconfig* to configure operating system.

21. Performs run-level specific initialization by executing startup scripts from */etc/init.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. For more details on services started at run level 3, please refer to the scripts in */etc/init.d/rc3.d.*

    - Saves and restores the system entropy tool for higher quality random number generation.
    - Starts the `atd` daemon.
    - Configures network interfaces.
    - Starts the system logging daemons.
    - Starts the `sshd` daemon.
    - Starts the `cron` daemon.
    - Probes hardware for setup and configuration.
    - Starts the program `agetty.`

The following diagram schematically describes the boot process.

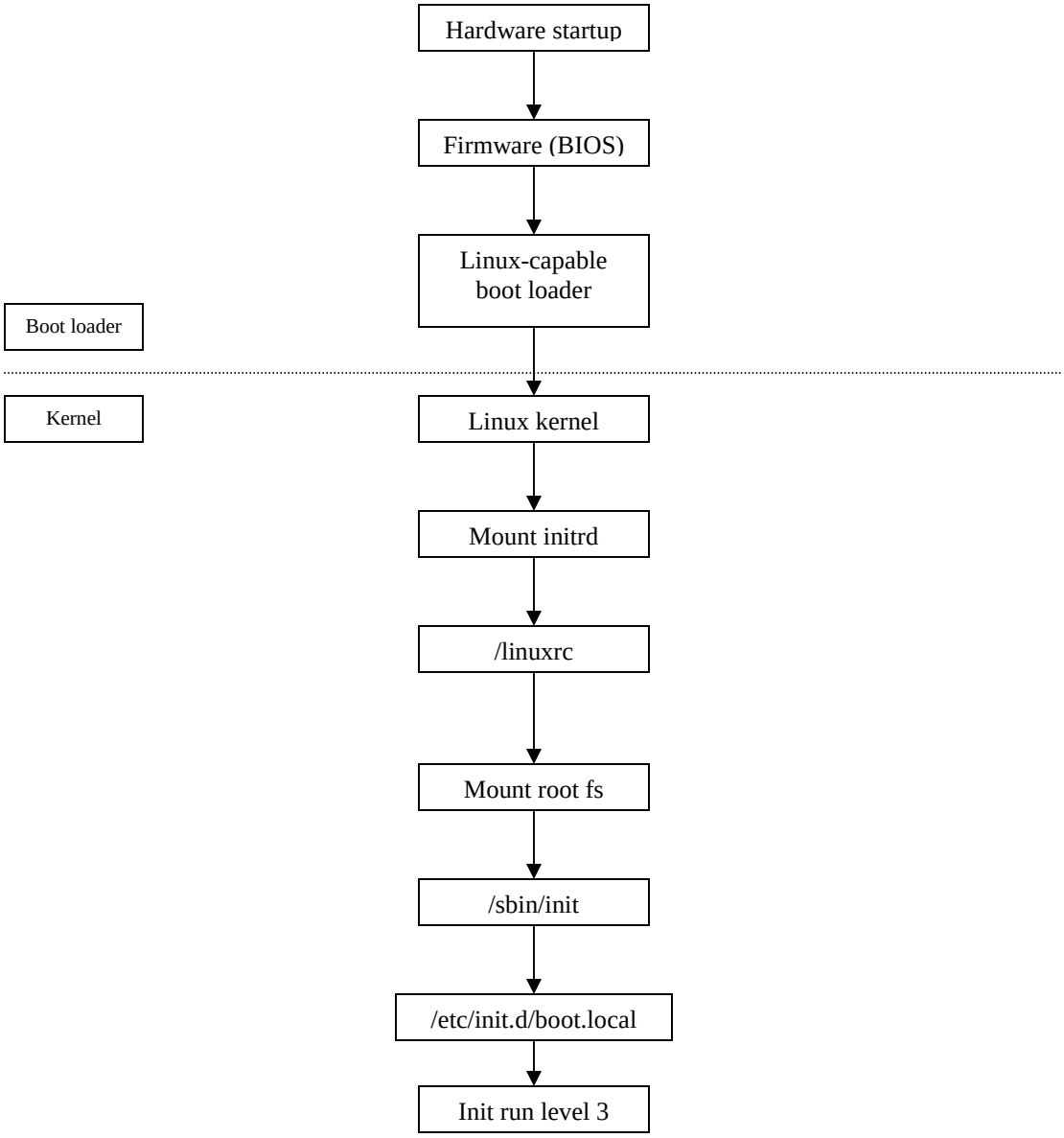

Figure 5-71. eServer 325 SLES boot sequence

5.10 Identification and authentication

Identification is when a user professes 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, 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. This section briefly describes PAM, protected databases and their functions, trusted programs and their high-level design implementation, and interaction of identification and authentication subsystem with audit. For more detailed information, please refer to the following:

Scott Mann, Ellen Mitchell and Michell Krell, Linux System Security, 2nd Edition
Kevin Fenzi, Dave Wreski, Linux Security HOWTO at *http://usr/share/doc/howto/en/html-single/Security-HOWTO.html*

5.10.1 Pluggable Authentication Modules

Pluggable Authentication Modules (PAM) is at the heart of 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 on PAM, please refer to the PAM project Web page at *http://www.kernel.org/pub/linux/libs/pam*.

5.10.1.1 Overview

PAM consists of a set of shared library modules that 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, groupdel
- sshd
- 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 no application-specific configuration file is found, then */etc/pam.d/other* is used.
3. Depending on the order specified in the configuration file, PAM loads the appropriate modules. Please refer to section 5.14 for the mechanics of loading a shared library.
4. The audit module opens the audit-device file, attaches the current process to the audit subsystem, and closes the audit device file.
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 audit module creates an audit record of type “Audit User Message” 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.

5.10.1.2 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 fields `module-type`, `control-flag`, `module-path`, and `arguments`.

`module-type`
Module types are `auth`, which tells the application to prompt the user for a password; `account`, which verifies various account parameters, such as password age; `session`, which performs pre- and post-processing of a session establishment; and `password`, which updates users’ authentication token.

`control-flag`
Control flags specify the action to be taken based on the result of a PAM module routine. When multiple modules are specified for an application (stacking), the control flag specifies the relative importance of modules in a 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.

`module_path`
Module path specifies the exact path name of the shared library module or just the name of the module in */lib/security*.

`arguments`
`The argument` field passes arguments or options to the PAM module. `arguments` can take values like `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.

5.10.1.3 Modules

SLES is configured to use the following PAM modules:

pam_unix2.so
Supports all four module types. `pam_unix2.so` provides standard password-based authentication. `pam_unix2.so` uses standard calls from the system's libraries to retrieve and set account information as well as to perform authentication. Authentication information on 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, 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_wheel.so
: Permits root access to members of the *trusted* group only. By default, `pam_wheel.so` permits root access to the system if the applicant user is a member of the *trusted* group (first, the module checks for the existence of a *trusted* group). Otherwise, the module defines the group with group ID 0 to be the *trusted* group. The TOE is configured with a *trusted* group of GID = 42.

pam_nolgin.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_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 specified in the */etc/login.defs* file.

pam_listfile.so
: Allows for the use of access control lists based on users, ttys, remote hosts, groups, and shells.

pam_deny.so
: Always returns a failure.

pam_laus.so
: Provides interfaces for processes to interact with the audit subsystem. Allows processes to attach to the audit subsystem and allows initialization of audit session for a user. Intercepts successes and failures reported by other PAM authentication modules, and generates appropriate user-space audit records for them.

5.10.2 Protected databases

The following databases are conferred by the identification and authentication subsystem during user session initiation:

/etc/passwd
: For all system users, stores 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 ":". The file is owned by user root and group root, 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 ":". The file is owned by user root and group root, and its mode is 644.

/etc/shadow

For all system users, stores 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 hashing algorithm MD5 is used to build the password checksum. The file is owned by user root and group shadow, and its mode is 400.

/var/log/lastlog

Stores the time and date of the last successful login for each user. The file is owned by user root and group tty, and its mode is 644.

/var/log/faillog

Stores the time and date of the last failed login attempt for each user. The file is owned by user root and group root, and its mode is 644.

/etc/login.defs

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 user root and group root, 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 user root and group root, and its mode is 644.

/var/run/utmp

The *utmp* file stores information about who is currently using the system. *utmp* 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(2)*. The file is owned by user root and group *tty*, 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 "~" with user name "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 user root and group tty, and its mode is 664.

/etc/ftpusers

The text file *ftpusers* contains a list of users who cannot log in using the File Transfer Protocol (FTP) server daemon. The file is owned by user root and group root, and its mode is 644.

5.10.3 Trusted commands and trusted processes

The Identification and Authentication subsystem contains the `agetty` and `mingetty` trusted processes, and the `login`, `passwd` and `su` trusted commands.

agetty

`agetty`, the alternative linux `getty`, is invoked from `/sbin/init` when the system transitions from a single-user mode to a multiuser mode. `agetty` opens a tty port, prompts for a login name, and invokes */bin/login* to authenticate. Please 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. For example, modem or regular terminal.
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.

mingetty

`mingetty`, the minimal linux `getty`, is invoked from `/sbin/init` when the system transitions from single-user mode to multiuser mode. `mingetty` opens a pseudo tty port, prompts for a login name, and invokes */bin/login* to authenticate. Please 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.

login

`login` is used when a user signs 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*. `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 in the */etc/login.defs* file. 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. Please 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 tty name.
4. Sets 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 password is required and username is not set yet, prompts for user name.
8. 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, exits.
10. Performs account management by calling `pam_acct_mgmt()`.
11. Sets up supplementary group list.
12. Updates `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 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.

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 utilizes 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. Please 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 history and password strength checks, and to update password. Generates audit record indicating successful update of the password.

su

`su` allows a user to switch identity. `su` changes the effective and real user and group ID to those of the new user. Please 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 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, 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.

5.10.4 Interaction with audit

Trusted processes and trusted commands of the identification and authentication subsystem are responsible for setting a process's credentials. 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 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's task structure to contain a login ID. This login ID can only be set once, and once set cannot be changed irrespective of process privileges. It is set by trusted processes and trusted programs that perform authentication. Programs such as login, crond, atd, 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 as follows:

Upon successful authentication, the audit module generates an audit record of type "Audit Login Message." The generation of login message results in invocation of `audit_login()` routine in the kernel. The `audit_login()` routine sets the login ID of the process. The login ID is not affected by calls such as `setuid()`, `setreuid()` and `seteuid()`, that are invoked by trusted command `su`. This login ID is contained in each audit record generated by the process, allowing all actions to be irrefutably traced back to the login user.

5.11 Network applications

This section describes the network applications subsystem. The network applications subsystem contains Secure Socket Layer (SSL) interface, `sshd` and `vsftpd` trusted processes, which interact with the PAM modules to perform authentication. The network application subsystem also includes the `xinetd` super-server and the `ping` program. These trusted processes and trusted programs recognize different hosts in the LAN with their IP addresses or with their names. Host names are associated with IP addresses using the */etc/hosts* file.

5.11.1 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 publicly available implementation of the Secure Socket Layer (SSL). SSL, which is encryption based, is a technology that provides message encryption, server authentication, message integrity, and optional client authentication. The section briefly describes the SSL protocol and how it is used to provide secure communication to and from a SLES system. For more detailed information on SSL, please refer to the following:

OpenSSL project Web site at http://www.openssl.org/docs

William Stallings, *Cryptography and Network Security Principles and Practice*, 2nd Edition

Adolfo Rodriguez, et al., IBM Redbook *TCP/IP Tutorial and Technical Overview*
http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf

Eric Young, *Internet Security Protocols: SSLeay & TLS*

Tim Dierks, Eric Rescorla, *The TLS Protocol version 1.1*
http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-05.txt

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.

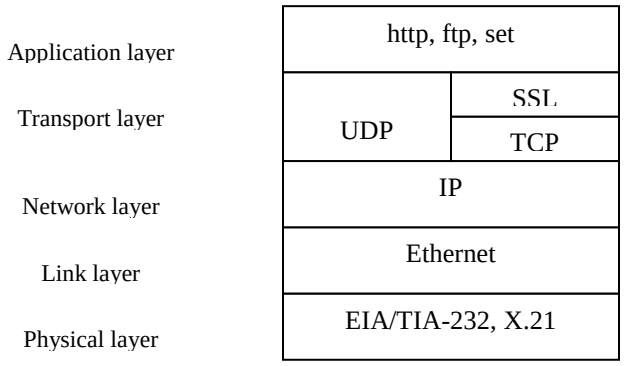

Figure 5-72. SSL location in the network stack

5.11.1.1 SSL concepts

At the heart of the SSL architecture is the use of 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.

Encryption
Encryption is a process of disguising a message. Encryption transforms a clear-text message into ciphertext.

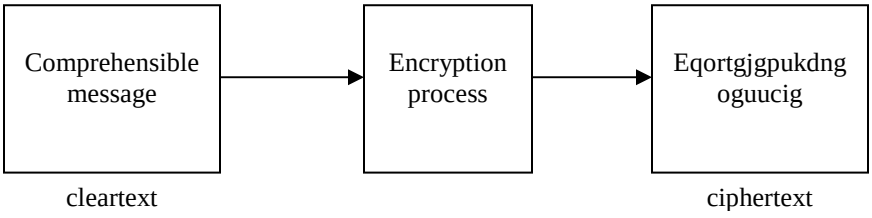

Figure 5-73. Encryption

Decryption converts ciphertext back into the original, comprehensible cleartext.

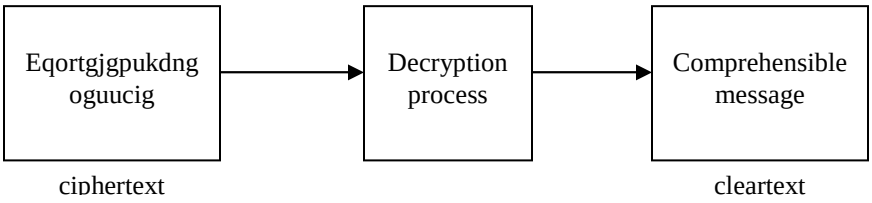

Figure 5-74. Decryption

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.

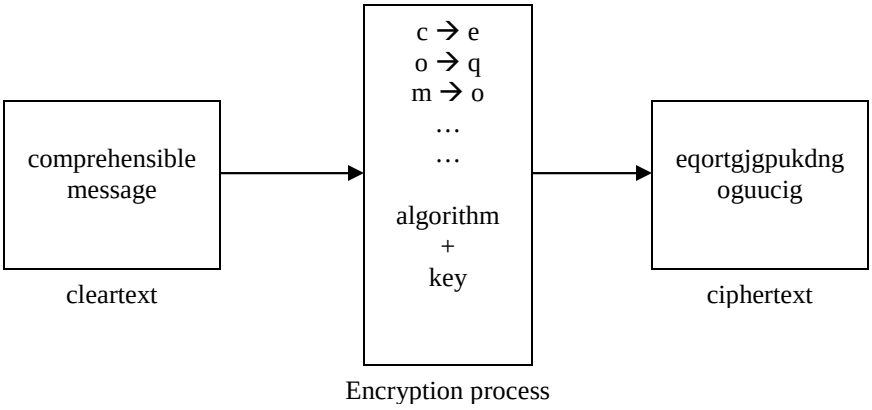

Figure 5-75. Encryption Algorithm and Key

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.

Encryption with symmetric keys

A symmetric key, also known as 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.

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.

The following diagram conceptually illustrates the creation of asymmetric keys for encryption and decryption.

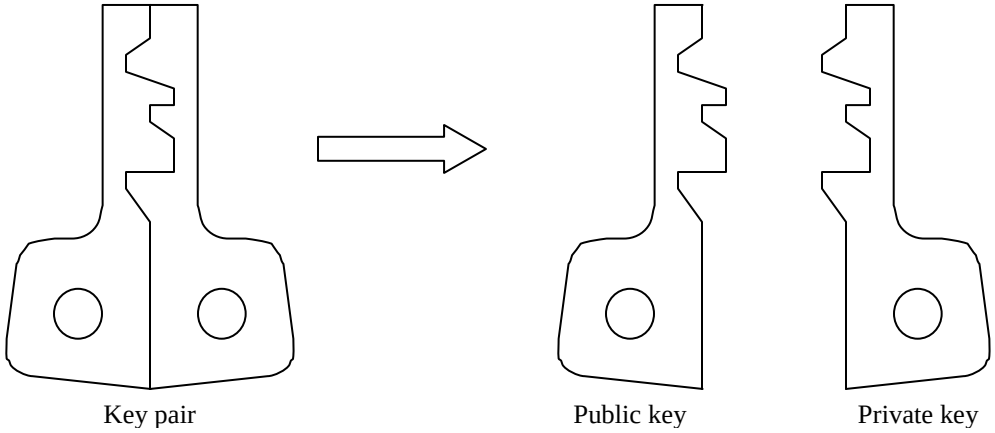

Figure 5-76. Asymmetric keys

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.

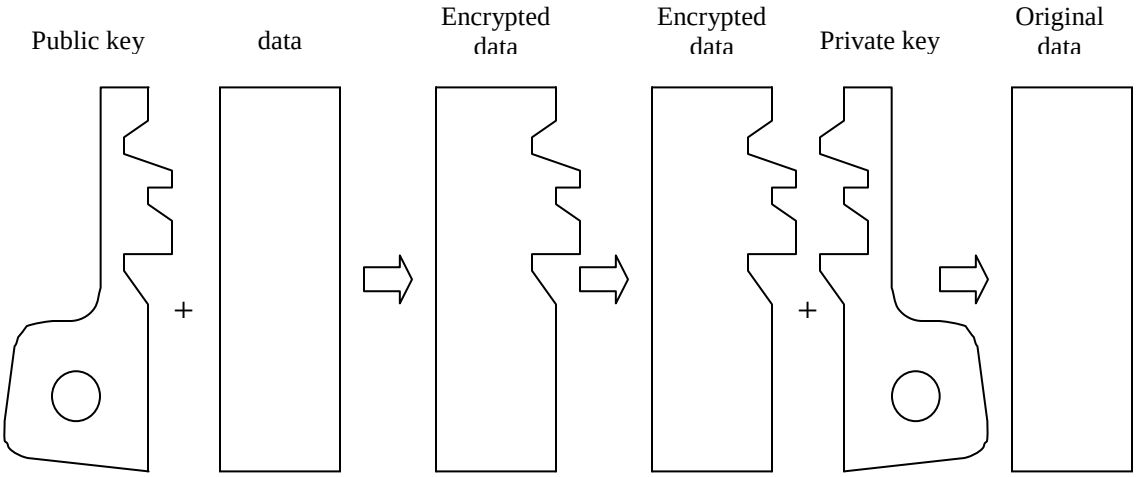

Figure 5-77. Encryption with public key provides confidentiality

If encryption is done with a private key, only the corresponding public key can be used for decryption. This gives the receiver of the message the ability to authenticate the sender. Encryptions of a message with the sender's private key acts like a digital signature, because only the corresponding public key of the sender can decrypt the message. Thus, indicating that the message was indeed sent by the sender. The following diagram conceptually illustrates the use of encryption with a private key to provide authentication.

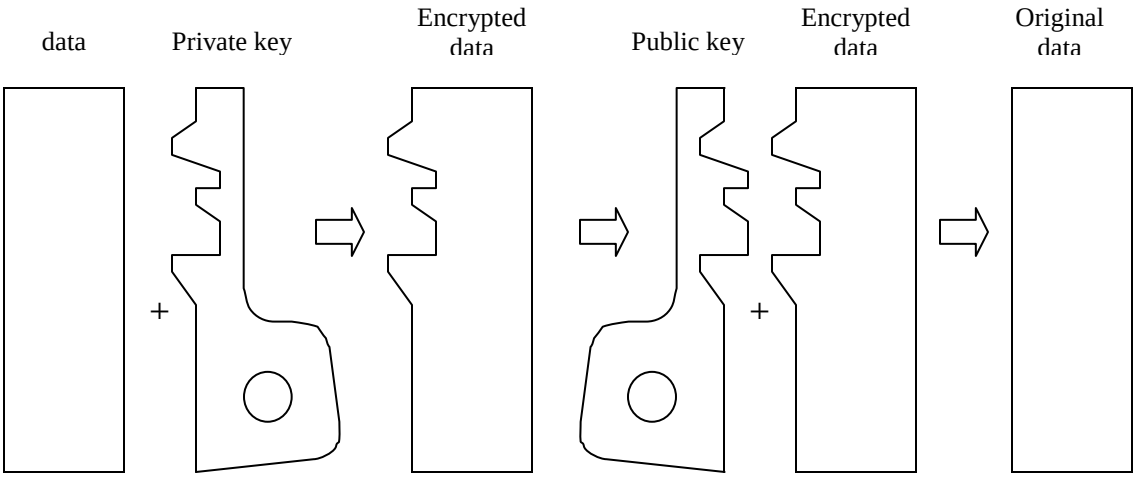

Figure 5-78. Encryption with private key provides authentication

Message digest

A message digest is 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 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 message 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.

Message Authentication Code (MAC)

A message authentication code (MAC) is a type of message digest that is created by encrypting, with a symmetric key, the output of a one-way hash function.

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 untrusted network, then that process introduces a security vulnerability. An intruder can intercept messages between them, replace their public keys with his own public key, 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.

5.11.1.2 SSL architecture

SSL occupies a space between the transport and application layer in the network stack. The SSL protocol itself 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. 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 (encryption and hash algorithms) used by the connection. The SSL Alert Protocol reports SSL-related errors to communicating peers.

The following diagram depicts different SSL protocols and their relative positions in the network stack.

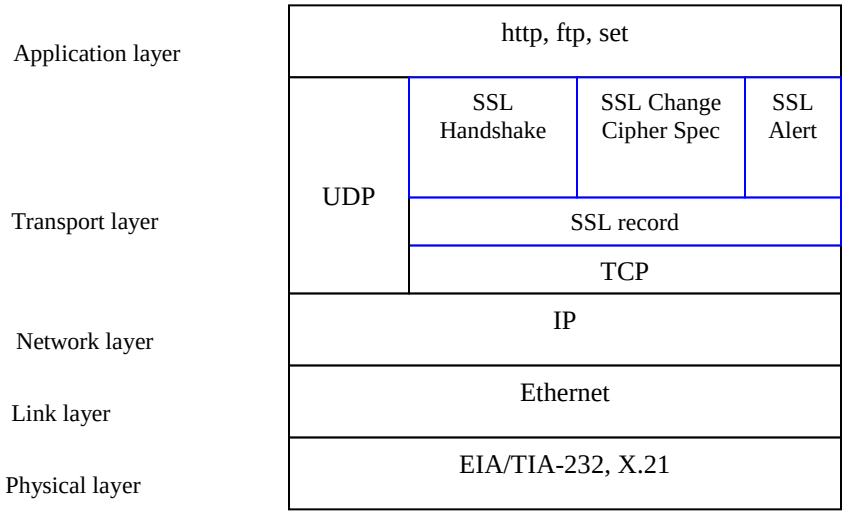

Figure 5-79. SSL Protocol

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 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.

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 (encryption algorithm, MAC algorithm, cryptographic keys) takes place. Security parameters, set up by the

handshake protocol, are used for all connections in a session. The following diagram from [STALLINGS] illustrates the handshake protocol. Additional details on actions taken at different stages of the handshake are provided on the next page.

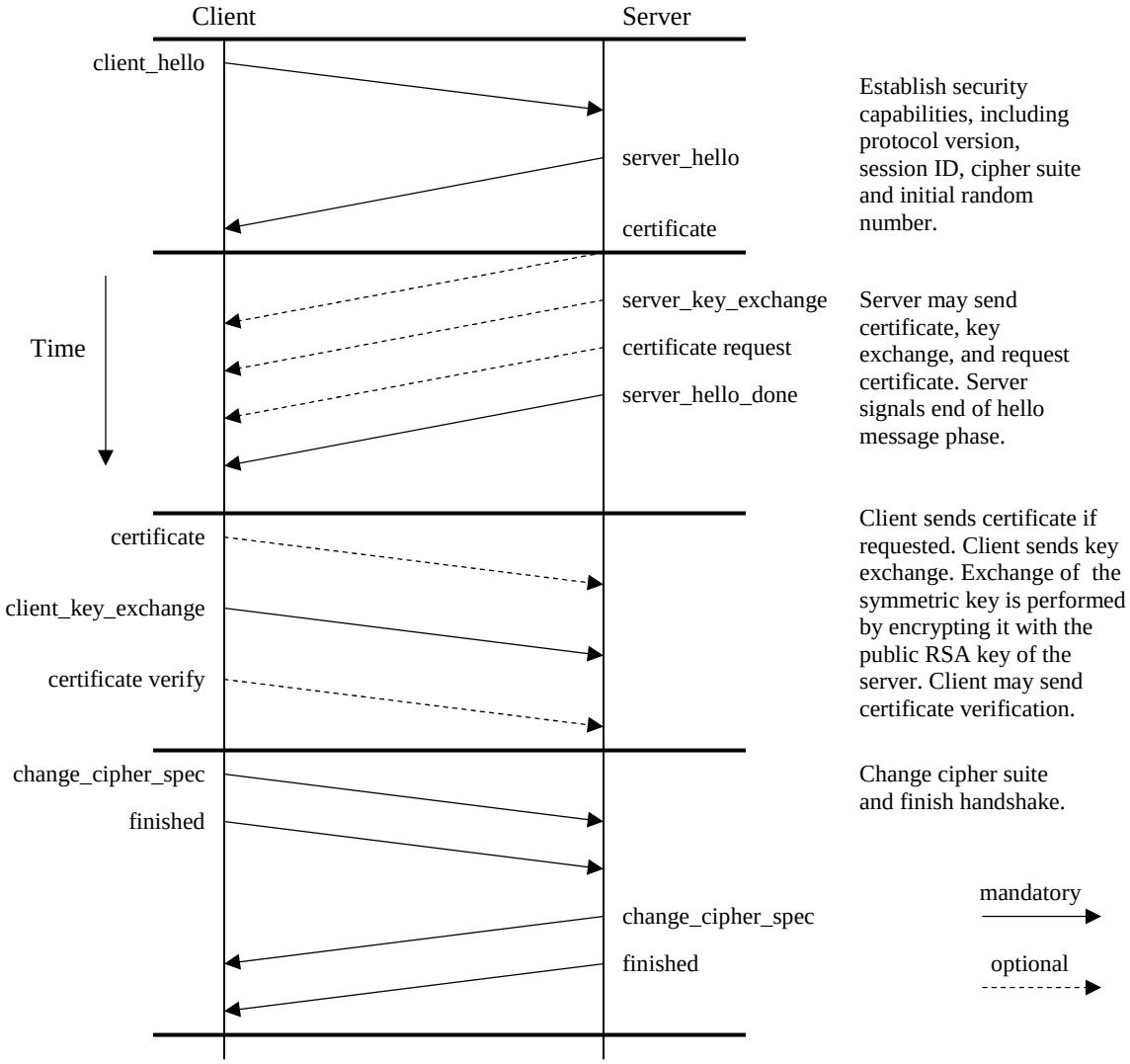

Figure 5-80. Handshake protocol action

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.

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 (e.g., DSS [DSS] certificates, signing-only RSA [RSA] certificates), or FORTEZZA KEA key exchange is used. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.

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.

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 (i.e. all certificates except those containing fixed Diffie-Hellman parameters).

Cipher suites supported

The evaluated configuration supports the following cipher suite:

CipherSuite	Key Exchange	Cipher	Hash
SSL_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA-1
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	TripleDES	SHA-1
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	AES (128 bit)	SHA-1
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	AES (256 bit)	SHA-1

Note: The last two cipher suites are defined in the IETF RFC 3268 for both the SSLv3 and the TLSv1 standard. In SLES9 they are supported by the implementation of SSLv3 in the OpenSSL library.

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.

SSL alert protocol

The SSL alert protocol communicates SSL-specific errors (for example, errors encountered during handshake or message verification) to the appropriate peer.

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 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 message authentication code (MAC) to the message. The MAC is created with another shared secret key negotiated by the handshake protocol.

The following diagram from [STALLINGS] depicts the operation of the SSL record protocol.

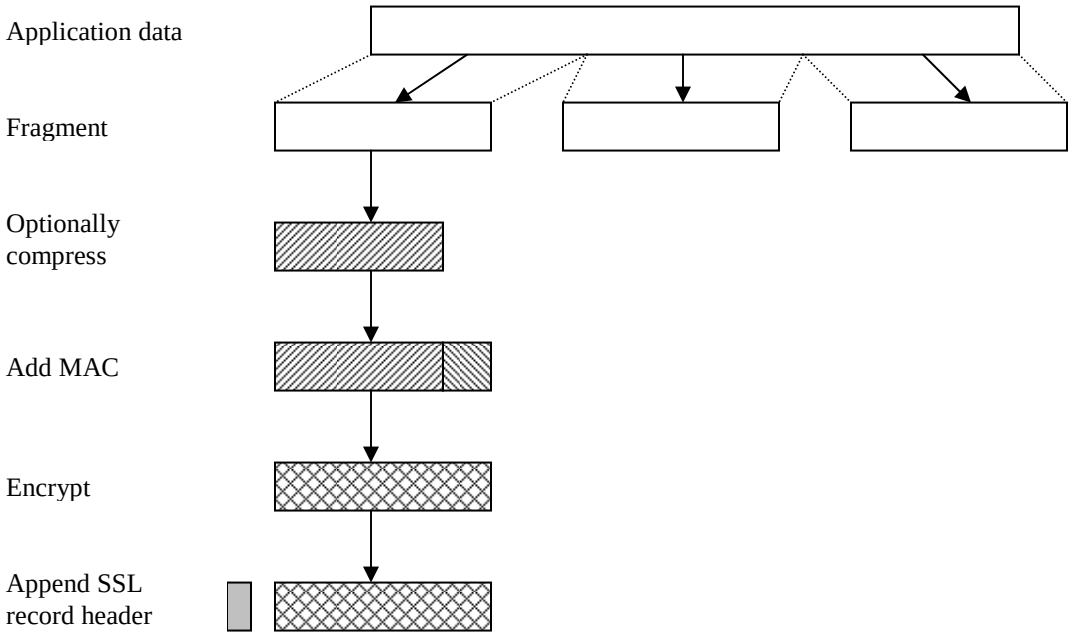

Figure 5-81. SSL record protocol operations

5.11.1.3 OpenSSL algorithms

This section briefly describes various encryption and hash algorithms supported by OpenSSL on TOE.

5.11.1.3.1 Symmetric ciphers

OpenSSL on TOE supports the following symmetric key encryption algorithms. For a detailed description of each of these algorithms, please refer to their manual 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.

DES
DES, or Data Encryption Standard, 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 three times with each encryption using a different key.

IDEA
The International Data Encryption Algorithm (IDEA) cipher is secret key block encryption algorithm developed by James Massey and Xuejia Lai. IDEA operates on 64-bit plaintext blocks and uses a 128-bit key.

RC4
RC4, proprietary of RSA Security Inc., is a stream cipher with 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.

AES
AES is the new encryption standard published by NIST in FIPS 197. It is a symmetric block cipher with a data block size of 128 bit and key sizes of 128, 192 and 256 bit.

5.11.1.3.2 Asymmetric ciphers

OpenSSL on TOE supports the following asymmetric key encryption algorithms. For a detailed description of each of these algorithms, please refer to their manual pages.

DSA
DSA (Digital Signature Algorithm) 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, 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.

5.11.1.3.3 Certificates

OpenSSL on TOE supports the following certificate format. For a detailed description of this format, please refer to its manual page.

X.509
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.

5.11.1.3.4 Hash functions

OpenSSL on TOE supports the following hash functions to generate message authentication codes (MAC). For a detailed description of each of these functions, please refer to their manual pages.

MD2, MD4 & MD5
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
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
RIPEMD is a cryptographic hash function with 160-bit output.

SHA-1
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.

5.11.2 ssh

ssh allows a user to run commands as if they are logged in on a text console of a remote system. 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, and session maintenance. Password expiration and locking are handled through the appropriate PAM functions.

The communication between the `ssh` client and ssh server uses the SSH protocol, version 2.0. SSH protocol requires that each host has 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 keyword "`ciphers`". 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 keyword "`MACs`". Please refer to section 5.11.1.3.1 for brief descriptions of these algorithms. Encryption is provided by the OpenSSL package, which is a separate package maintained by an independent group of developers. 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 (for example, there is only one initialization vector). This is a block cipher with 8 byte blocks. This algorithm is defined in [SCHNEIER].

Integrity check
Data integrity is protected by including with each packet a message authentication code (MAC) 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/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` singed 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 [SCHNEIER].

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.

The following paragraphs briefly describe the implementation of the *ssh* client and the *ssh* server. For detailed information on the SSH Transport Layer Protocol, SSH Authentication Protocol, SSH Connection Protocol and SSH Protocol Architecture, please refer to the corresponding protocol documents at the Web site *http://www.ietf.org/internet-drafts*.

5.11.2.1 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 the user is typing his or her password. *ssh* prevents the password from being displayed on the terminal as it is being typed. The *ssh* client then makes any pty and forwarding requests and can call code *in ttymodes.c* to encode current tty modes. Finally, it 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 (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, the *ssh* client is not SUID root on the system.

5.11.2.2 ssh server (sshd)

The `sshd` daemon starts by processing arguments and reading the configuration file */etc/ssh/sshd_config*. The configuration file contains keyword-argument pairs, one per line. Please refer to the *sshd_config(5)* manual page for available configuration options. It then reads the host key, starts listening for connections, and generates the server key. The server key is regenerated every hour by an alarm.

When the server receives a connection, it forks, disables the regeneration alarm, and starts communicating with the client. The server and client first perform identification string exchange, 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*.

5.11.3 xinetd

`xinetd` is the super server that starts other servers that provide network services, such as file transfer, between systems. `xinetd` starts at system initialization and listens on all service ports for the services

listed in its configuration file */etc/xinetd.conf*. When a request comes in, *xinetd* starts the appropriate server. The *xinetd* super server conserves system resources by avoiding having to fork a lot of processes that might be dormant for most of their lifetime. `xinetd` also provides access control and logging. The remainder of this section describes some of the security-relevant features of `xinetd`. For additional information on `xinetd` and its configuration, please refer to the following:

Scott Mann, Ellen Mitchell and Michell Krell, Linux System Security, 2nd Edition, Chapter 10
Hal Burgiss, Security-QuickStart HOWTO for Linux at *http://usr/share/doc/howto/en/html-single/Security-QuickStart-HOWTO.html*
http://www.xinetd.org

`xinetd` provides the following security-relevant features:

- Provides access control for TCP, UDP, and RPC services.
- Provides access limitations based on time.
- Provides for killing of services that are no longer allowed.
- Limits the number of daemons of a given type that can run concurrently, which helps prevent Denial of Service (DoS) attacks.
- Limits overall number of processes forked by `xinetd`, which helps prevent DoS attacks.
- Provides extensive logging capabilities for both successful and unsuccessful connections.
- Limits log file sizes, which helps prevent DoS attacks.

Network services and their behaviors are configured through the */etc/xinetd.conf* configuration file. Each entry in */etc/xinetd.conf* is of the following form:

```
Service service_name
{
attribute operator value value …
…
…
}
```

`service` is a required keyword and the braces surround the list of attributes. The `service_name` is arbitrary, but is chosen to conform to the standard network services in the default SLES configuration. Attributes used for security relevant configurations are:

`access_times`
: Sets the time intervals for when the service is available. The format is `hh:mm-hh:mm`. Hours range from 0 to 23 and minutes can be from 0 to 59.

`only_from`
: Space-separated list of allowed client systems in resolvable names or IP addresses. If this attribute is specified without a value, it acts to deny access to the service.

`no_access`
: Space-separated list of denied clients in resolvable names or IP addresses.

`instances`
: Accepts integer greater than, or equal to, one or `UNLIMITED`. Sets the maximum number of concurrent running daemons.

`per_source`
: Accepts integer or `UNLIMITED`. Specifies the maximum number of instances of a service per source IP address.

`cps`

Accepts two arguments: number of connections per second to handle, and number of seconds to wait before reenabling the service if it has been disabled. `cps` limits the rate of incoming connections.

`max_load`

Accepts a floating point value, which is the load at which the service will stop accepting connections.

5.11.4 vsftpd

`vsftpd` is the Very Secure File Transfer Protocol daemon. `vsftpd` provides a secure, fast, and stable file transfer service to and from a remote host. `vsftpd` is invoked from the *xinetd* super-server. The behavior of `vsftpd` behavior can be controlled by its configuration file */etc/vsftpd.conf*. The remainder of this section describes some of the security-relevant features of `vsftpd`. For additional information on `vsftpd` and its configuration, please refer to the following:

*/usr/share/doc/packages/vsftpd/SECURITY/**
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 directory.
- Ability to hide all user and group information in directory listing.
- Ability to set up secure tunneling scheme.
- Ability to perform enhanced logging.
- Ability to set up up connection timeout values.

The daemon generally follows these steps:

1. Parses command line arguments.
2. Parses configuration file.
3. Performs sanity checks such as ensuring that standard input is a socket.
4. Initializes the session.
5. Sets up environment.
6. Starts up logging.
7. Depending on the configuration, start one or multiple process session.
8. Invokes appropriate functions to initiate connection.
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.
    a. Invokes `pam_start()` to initialize the PAM library and to identify the application with a particular service name.
    b. Invokes `pam_authenticate()` to authenticate the application user. Terminal echo is turned off while the user is typing his/her password.
    c. Invokes `pam_acct_mgmt()` to perform module specific account management.
    d. Invokes `pam_setcred()` to set credentials.
    e. Invokes `pam_end()`.

5.11.5 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. ECHO_REQUEST datagrams (``pings") have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of “pad” bytes used to fill out the packet.

5.11.6 openssl

`openssl` is a command line interface to the OpenSSL cryptography toolkit, which implements the Secure Socket Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them. 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/TLS client and server tests.
- Handling of S/MIME signed or encrypted mail.

For more detailed information on the `openssl` command and its usage, please refer to the following `openssl` man page at the *openssl.org* Web site.

http://www.openssl.org/docs/apps/openssl.html

5.11.7 stunnel

`stunnel` is designed to work as an SSL encryption wrapper between remote clients and local, xinetd startable, or remote servers. `stunnel` can be used to add SSL functionality to commonly used xinetd 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 connect option, or a new program via the exec option. There is also an option to allow a program to accept incoming connections and then launch `stunnel`, for example with `xinetd`.

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 file */etc/stunnel.pem*. `stunnel` uses the openssl library and therefore can use the cipher suites implemented by that library. The SSL cipher suites supported in the evaluated configuration are SSL_RSA_WITH_RC4_128_SHA, SSL_RSA_WITH_3DES_EDE_CBC_SHA, TLS_RSA_WITH_AES_128_CBC_SHA and TLS_RSA_WITH_AES_256_CBC_SHA.

Stunnel is configured by the file */etc/stunnel/stunnel.conf*. 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 sqare 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. For example,

```
# Global parameters
cert = /usr/local/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 on stunnel, please refer to its man page as well as the following links:

http://stunnel.mirt.net
http://www.stunnel.org

5.12 System management

This subsystem contains the trusted programs used for system management activities. They include `chage`, `chsh`, `chfn`, `useradd`, `usermod`, `userdel`, `groupadd`, `groupmod`, `groupdel`, `gpasswd`, `date`, and `amtu`.

chage

`chage` allows the system administrator to alter a user's password expiration data. Please refer to the `chage` man page for more detailed 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 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.

chsh

`chsh` allows a user to change his or her login shell. If a shell is not given on the command line, `chsh` prompts for one. Please refer to the `chsh` man page for detailed information on usage of the command. `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 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. Invokes `pam_setcred()` to set credentials.
11. Prompts for new shell if one is not provided on the command line.
12. Checks the shell to make sure that it is accessible.
13. Invokes `setpwnam()` to update appropriate database files with the new shell.
14. Exits.

chfn

`chfn` allows a user to change his or her finger information. The information, stored in */etc/passwd* file, is displayed by the `finger` command. Please refer to the `chfn` man page for detailed information on usage of the command. `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 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.

useradd

`useradd` allows an authorized user to create new user accounts on the system. Please refer to the `useradd` man page for more detailed information on usage of the command. `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 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. 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.

usermod

`usermod` allows an administrator to modify an existing user account. Please 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 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. Actions, such as 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.

userdel

`userdel` allows an administrator to delete an existing user account. Please refer to the `userdel` man page for more detailed information on the usage of the command*.`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 created by the user.
15. Invokes `pam_chauthok()` to rejuvenate the user's authentication tokens.
16. Exits.

groupadd

`groupadd` allows an administrator to create new groups on the system. Please 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 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 group being created doesn't exist already.
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.

groupmod

`groupmod` allows an administrator to modify existing groups on the system. Please refer to the `groupmod` man page for more detailed information on usage of the command. `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 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 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 user's authentication tokens.
12. Exits.

groupdel

`groupdel` allows an administrator to delete existing groups on the system. Please refer to the `groupdel` man page for more detailed information on usage of the command. `groupdel` generally follows these steps:

1. Sets language.
2. Invokes `getpwuid( getuid())`to obtain the application user's `passwd` structure.
3. Invokse `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 group being deleted does exist, and that it is not the primary group for any users.
8. Invokes `open_files()`to lock and open authentication database files.
9. Invokes `grp_update()` to update group information. Generates audit record to log deletion of existing groups.
10. Invokes `close_files()`to close the authentication database files.
11. Invokes `pam_chauthok()`to rejuvenate the user's authentication tokens.
12. Exits.

date

`date`, for a normal user, displays current date and time. For an administrative user, `date` can display or set the current system time. Please refer to the `date` man page for detailed information on usage of the command. `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 `stime()`system call to set 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 to the command.
5. Process return from the `stime()`system call. Print current time or error depending on the return value from the system call.
6. Exits.

gpasswd

`gpasswd` administers the */etc/group* and */etc/gshadow* files. `gpasswd` allows system administrators to designate group administrators for a particular group. Please refer to the `gpasswd` man page for more detailed information. Group passwords are not used on the TOE.

amtu (abstract machine test utility)

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*, which is providing functionalities 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, please refer to the following:

Emily Ratliff, *Abstract Machine Testing: Requirements and Design*

The test tool performs the following tests:

Memory
The tool allocates 10% of the free memory of the system, 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. If the allocation of 10% of the free memory fails, the tool performs the above test after allocating 5% of the free memory.

Memory separation
To fulfill the memory separation requirement, the test tool performs the following:

1. As a normal user, the tool picks random areas of memory in ranges reported in */proc/self/maps* 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.

I/O controller - network
Because portions of the TSF depend on the reliability of the network devices and the disk controllers, the test tool 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. The tool performs the following:

1. Using the PF_PACKET communication domain, opens another connection to the listening server.
2. Ensures that the random data transmitted is also the data received.

These steps are repeated for each configured network device.

I/O controller – disk
In order to check the disk controllers (IDE and SCSI only), the test tool 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 string size 10 MB is chosen so that the file exceeds the size of the device's buffer. The AMTU utility 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.)

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 should be 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. Instructions to do this for each architecture are given below.

pSeries, iSeries

The instruction set for the PowerPC® processor is given in the book at the following Web address:

http://www-3.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. The following instructions are tested by this tool:

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).

zSeries

Principles of Operation is a handy reference for the zSeries 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).

xSeries

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 for the x86 architecture.

The following privileged instructions are tested by this tool:

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: MOVL %cs,28(%esp) – Overwrite 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).

eServer 325

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 a list of privileged instructions for the AMD64 architecture.

The following privileged instructions are tested by this tool:

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: MOVL %cs,28(%esp) – Overwrite 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).

Utility output
For each of these subsystems, the 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 Batch processing

On an SLES system, a user can submit jobs in the background for execution at a later time. 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 user commands `at`, `batch`, and `crontab`, and trusted processes `atd` and `cron`. The command `batch` is a script that invokes `at`; hence, only `at` internals are described in this section.

5.13.1 Batch processing user commands

Batch processing user commands are `at` and `crontab`. `at` schedules the one-time execution of jobs based on time value; `crontab` uses a control file to dictate when repeated jobs will execute.

5.13.1.1 at, atrm

`at` reads commands from standard input and sets them up to be executed at a later time. `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. If */etc/at.allow* exists, only users listed in that file are allowed to use this command. If */etc/at.deny* exists, any users listed in the file are not allowed to use this 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 on 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.

5.13.1.2 crontab

`crontab` allows an administrator to perform specific tasks on a regularly scheduled basis without logging in. A user can create a *crontab* file with the help of this command. *crontab* files are processed by the `cron` trusted process daemon. `crontab` 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 */var/spool/cron/allow* file exists only users listed in that file are allowed to use this command. If the */var/spool/cron/deny* file exists, users listed in that file are not allowed to use this command. Generates an audit record if a user is not allowed to use this command.
3. If listing *crontab*, invokes the `list_cmd()` routine to list the existing *crontab* file. 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. Generates audit record to log the deletion of an existing *crontab* file.
5. If editing *crontab*, invokes the `edit_cmd()` routine to edit the existing *crontab* file. Generates audit record to log modification of an existing *crontab* file.
6. If replacing *crontab*, invokes the `replace_cmd()` routine to replace the existing *crontab* file. After both the edit and replace options, the command ensure that the modified/new *crontab* file is owned by root and has an access mode of 600. Generates audit record to log the replacement of an existing *crontab* file.

crontab files are created in the */var/spool/cron/tabs* directory and are created with the login name of the respective user. The `cron` daemon uses the name of the file to determine 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.

5.13.2 Batch processing daemons

Trusted processes that implement batch processing are `atd` and `cron`. `atd` runs jobs queued by the `at` command; `cron` executes commands scheduled through *crontab* or listed in */etc/crontab* for standard system `cron` jobs.

5.13.2.1 atd

`atd` is the trusted process daemon that services users' requests for timed execution of specific tasks. `atd` ensures that the system's discretionary access control policy is not violated by exactly duplicating the identity for the user on whose behalf it is performing tasks. `atd` depends on the trusted command `at` to have appropriately created an `at` jobs file containing pertinent information about the user's identity. `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.

5.13.2.2 cron

`cron` is the trusted process daemon that processes users' *crontab* files. `cron` ensures that the system's discretionary access control policy is not violated by duplicating the login environment of the user whose *crontab* file is being processed. `cron` depends on the trusted command `crontab` to create the *crontab* file of each user with his or her name. The file */var/spool/cron/tabs/root* contains the crontab for root and; therefore, is highly critical. `cron` also depends on the kernel's file system subsystem to prevent normal users from creating or modifying other users' *crontab* files. `cron` is started 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 up system's `cron` jobs by reading *crontab* files in the directory */etc/cron.d/*.
3. Sets up `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 routine `load_database()` to read *crontab* files existing 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 up 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. Generates audit records to log execution of `cron` jobs.

5.14 User level audit subsystem

This subsystem contains the portion of the audit system that lies outside the kernel. This subsystem contains `auditd` trusted process, which reads audit records from kernel buffer and transfer them to on-disk audit logs, trusted audit management utilities `aucat`, `augrep`, `aurun` and `audbin`, audit logs, audit configuration files, and audit libraries.

5.14.1 Audit daemon

The audit daemon reads audit records from the kernel buffer through the */dev/audit* device and writes them to disk. The audit daemon supports three modes for writing records to disk. In *file* mode, data is written the same way as it is in `syslogd`; that is, records are appended to a file that is allowed grow arbitrarily. *stream* mode is similar to the file mode, except that data is sent to an external command on standard input. This allows forwarding of audit data to other commands or hosts. In *bin* mode, arbitrary numbers of fixed length files are maintained with a pointer to current location. The audit records are written until the current file has reached its maximum capacity and then the next file is utilized until it reaches its maximum capacity. The files are used in a round-robin fashion.

In addition to writing audit records to disk, the audit daemon sends configuration parameters, such as filter policy, to the kernel, turns kernel auditing on and off, and monitors the current state of the system for potential audit record loss. The audit daemon performs the following steps:

- Parses command line arguments.
- Parses filter configuration file.
- Sets up output file(s).
- Configures disk space thresholds.
- Becomes a daemon (run in background) and sets up the signal handler.
- Opens audit device (/dev/audit) and clears the filter policy for each system call.
- Exports policy initialized from the filter configuration file to the kernel.
- In an infinite loop, reads from audit device (kernel buffer) and writes the audit record to output file.
- Before each write, checks the disk-space threshold. If the disk space of the file system containing the output file exceeds the threshold value, generates an alarm. If the file system becomes full, the kernel suspends the processes attached to the audit subsystem to prevent loss of any audit data.

5.14.2 Audit utilities

The user space utilities consist of `aucat`, `augrep`, `aurun`, and `audbin`.

`aucat`

`aucat` reads the binary audit log files and outputs the records in human readable format. `aucat` supports ASCII format. The `aucat` command performs the following steps:

- Parses command line arguments.
- Sets up output format based on the command line argument.
- Invokes the `audit_print()` function of the audit server API library to print each and every audit record in the audit log.

`augrep`

`augrep` performs a similar function as `aucat`, but it allows an administrative user to optionally filter the records based on user, audit ID, outcome, system call, or file name. `augrep` supports ASCII format. The `augrep` command performs the following steps:

- Parses command line arguments.
- Sets up filter options based on command line arguments.
- Invokes the `audit_process_log()` function of the audit server API library to process and print audit records based on the filter options.

Filter options allow the selection of audit records based on the following criteria:

- User ID – effective, real and file system
- Group ID – effective, real and file system
- Login ID
- System call with specified outcome (success/failure)
- Process ID
- Event type
- Netlink message with specifiec group, dstgroup or result
- Login from specific remote hostname, host address or executable name
- Start time / End time
- Audit ID
- Exit message with specified exit code

- Text message

aurun

aurun is a wrapper application that allows attachment of trusted processes, such as Web servers, to the audit subsystem. The aurun command performs the following steps:

- Parses command line arguments.
- If a user is given on the command line, verifies that it is valid user. Initialize user's groups.
- Opens the audit subsystem and attach to it.
- Closes the audit subsystem and sets the UID to that given on the command line.
- Executes the program specified on the command line.

audbin

audbin is a trusted application that manages audit log files. audbin archives files that are generated when the system is running in bin mode. audbin parses command-line arguments and, if requested, clears the log file after saving its contents. audbin supports appending or overwriting the existing audit log file.

5.14.3 Audit logs

LAuS 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 LAuS audit utilities, such as aucat and augrep, on audit logs to extract and analyze security-relevant events.

Audit logs are protected by their discretionary-access control mod, in order to protect them from unauthorized deletion or modification.

Audit records in an audit log contain the major and minor version numbers of LAuS and a flag specifying the byte order. An audit record consists of a record header and a variable message. Each audit record contains information such as timestamp, login ID, audit ID and process ID along with variable audit data that depend on the type of the event. 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	LAuS event codes
Startup and shutdown of the audit functions.	AUDIT_start, AUDIT_stop
Modifications to audit configuration files.	Events AUDCONF_reload (generated by auditd); syscalls open, link, unlink, rename, truncate (write access to configuration files)
Reading of information from audit records	syscall open (on the audit log files)
Audit storage space exceeds a threshold	AUDIT_disklow
Audit storage space failure	AUDIT_diskfail
Operations on file system objects	syscalls chmod, chown, setxattr, link, mknod, open, rename, truncate, unlink, rmdir, mount, umount
Operations on message queues	syscalls msgctl and msgget
Operations on semaphores	syscalls semget, semctl, semop
Operations on shared memory segments	syscalls shmget, shmctl
Rejection or acceptance by the TSF of any tested secret.	Events AUTH_success, AUTH_failure (from PAM framework, ``authentication'' subtype)
Use of identification and authentication mechanism.	Events AUTH_success, AUTH_failure (from PAM framework, ``authentication'' subtype)

Event Description	LAuS event codes
Success and failure of binding user security attributes to a subject (e.g. success and failure to create a subject).	`LOGIN` audit record (from *pam_laus.so* module or `aurun`); syscalls `fork` and `clone`
All modifications of subject security values	syscalls `chmod, chown, setxattr, msgctl, semctl, shmctl`
Modifications of the default setting of permissive or restrictive rules.	syscalls `umask, open`
Modifications to TSF data.	syscalls `open, rename, link, unlink, truncate` (of audit log files and audit configuration files), AUDCONF_reload event, ``gpasswd'' audit text messages (from shadow suite), details include new value of of the TSF data
Modifications to the group of users that are part of a role.	Event: ``gpasswd:'' audit text messages ``group member added'', ``group member removed'', ``group administrators set'', ``group members set'' (from trusted programs in shadow suite).
Execution of the test of the underlying machine and the result of the test.	Event: ADMIN_amtu (generated by AMTU testing tool)
Changes to system time.	Event: syscalls settimeofday, adjtimex, stime
Setting up a trusted channel	Event: syscall exec (of stunnel program)

Table 5-2. Audit Subsystem event codes

5.14.4 Audit configuration files

The configuration file */etc/audit/audit.conf* is used to set the path to the filter rules, and to define the threshold for disk space. The configuration file */etc/audit/filter.conf* sets up filter rules. The configuration file */etc/audit/filesets.conf* contains a list of pathnames on which audit records could be filtered. Configuration files are simple ASCII files and can be changed with any text editor. Once the files are updated, the `` `auditd -r` `` can be used to notify the audit daemon of the new configuration files. The audit daemon then performs the appropriate `ioctl()`calls to load the new configuration parameters into the kernel as illustrated in the following diagram.

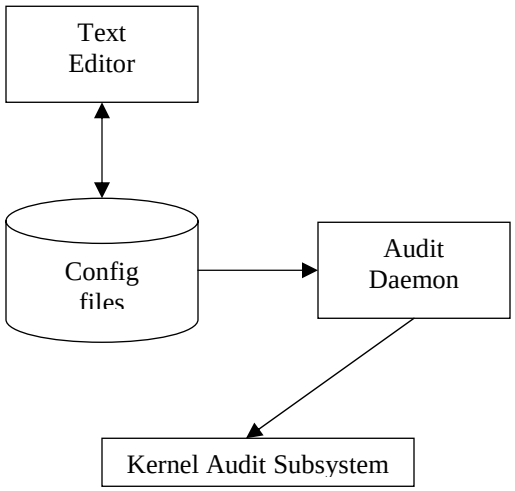

Figure 5-82. Audit subsystem configuration

5.14.5 Audit libraries

The audit subsystem provides two libraries: one for interacting with the kernel and the other for interacting with the audit daemon. The kernel API provides trusted program functions to attach to and detach from the audit subsystem, and functions to generate application audit records. Using a kernel API, a trusted program can generate a single, more descriptive, audit record instead of numerous audit records generated by each system call executed by the program.

The second library is used by applications to interact with the audit daemon. LAuS audit utilities use functions from this server API library to communicate configuration parameters, read audit logs and display them in human readable form, and send control messages.

5.15 Supporting functions

On a SLES system, all trusted programs and trusted processes 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.

5.15.1 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 discretionary access control mechanism.

Library	Description
/lib/libc.so.6	Runtime library of C functions.
/lib/libcrypt.so.1	Library that performs one-way encryption of user and group passwords.
/lib/libxcrypt.so	Replacement library for `libcrypt.so`. Supports bigcrypt and blowfish password encryption.
/lib/security/pam_unix2.so	Modules that perform basic password-based authentication, configured with the MD5 hashing algorithm.
/lib/security/pam_pwcheck.so	Modules that use *cracklib* to ensure stronger passwords.
/lib/security/pam_passwdqc.so	Modules that enforce additional stricter password rules. For example, reject passwords that follow keyboard patterns such as "1qaz2wsx".
/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_laus.so	LAuS PAM module to create audit records from authentication modules.
/lib/security/pam_tally.so	Modules that deny access based on the number of failed login attempts specified in the */etc/login.defs* file.
/lib/security/pam_listfile.so	Modules that allow use of access control lists based on users, ttys, remote hosts, groups, and shells.
/lib/security/pam_deny.so	Module that always returns a failure.

Library	Description
/lib/liblaus.1a	LAuS kernel application interface library for trusted commands to interact with the LAuS kernel component.
/lib/liblaussrv.1a	LAuS server application interface library for trusted commands to interact with the LAuS audit daemon.
/usr/lib/libssl3.so	OpenSSL library with interfaces to Secure Socket Layer version 3 and Transport Layer Security version 1 protocols.
/lib/libcrypto.so.2	OpenSSL crypto library with interfaces to wide range of cryptographic algorithms used in various Internet standards.

Table 5-3. TSF libraries

5.15.2 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's 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 executing process's address space. 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.

This mapping of page frames in a read-only text segment is carried out by the kernel 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 Discretionary Access Control requirement. Page frames used for this mapping are allocated with the demand paging technique, described in section 5.5.3, 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 variable LD_PRELOAD lists object files with functions that override the standard set. The variable LD_LIBRARY_PATH 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 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, the system presumes the program is setuid/setgid (or descended from one) and 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 Discretionary Access Control or object reuse issues with static libraries.

5.15.3 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 briefly describes the system call interrupt setup for the different hardware architectures that are part of the TOE.

5.15.3.1 xSeries

On xSeries systems, the Intel processors' Interrupt Descriptor Table is initialized to allow a trap gate that can be accessed by a user-mode process. This mapping is done at system initialization time by the routine `trap_init()`, which sets up the Interrupt Descriptor Table (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 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 `syscall_exit()` function. For a more detailed explanation of the system call invocation, please refer to the *SLES Low Level Design*, by Janak Desai, George Wilson, and Michael Halcrow.

5.15.3.2 pSeries and iSeries

On pSeries and iSeries, the PowerPC architecture provides the assembly instruction `sc` (supervisor call) to make a system call. The `sc` instruction is also used by the kernel 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, the `sc` instruction is treated as a hypervisor call. Otherwise, it is treated as 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 handler, `DoSyscall()`. `DoSyscall()`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_1()`function.

5.15.3.3 zSeries

On zSeries, z/Architecture provides the assembly instruction `SVC` (SuperVisor Call) 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 `sysc_return()`function.

5.15.3.4 eServer 325

The AMD Opteron processors differ significantly from x86 architecture with respect to the entry point into the kernel. The Opteron processor provides special instructions `SYSCALL` and `SYSRET` instead of using the interrupt 0x80. Assembly instruction `SYSCALL` performs a call to the system call handler `system_call()` running at CPL level 0. The address of the target procedure is specified implicitly through Model Specific Registers (MSR). `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.

5.15.4 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, the SLES kernel performs validation in a two-step process, as follows:

1. Verifies that the linear address (virtual address for iSeries, pSeries and zSeries) passed as a parameter does not fall within the range of interval addresses reserved for the kernel. That is, the linear address is lower than `PAGE_OFFSET`.
2. Since bad addresses lower than `PAGE_OFFSET` cause a page fault, 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.

The above satisfies the Access Control requirement.

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 5.10 provides details of the SLES system's Identification and Authentication subsystem.

6.1.1 User identification and authentication data management (IA.1)

Section 5.10.2 provides details of the configuration files for user and authentication management. Section 5.10.3 explains how a password can be changed.

6.1.2 Common authentication mechanism (IA.2)

Section 5.10.1 provides a description of PAM, which is used to implement the common authentication mechanism for all activities that create a user session.

6.1.3 Interactive login and related mechanisms (IA.3)

Section 5.10.3 provides a description of the interactive login process. Section 5.11.2 describes the process of obtaining a shell on a remote system.

6.1.4 User identity changing (IA.4)

Section 5.10.3 provides a description of changing identity on the local system using the *su* command.

6.1.5 Login processing (IA.5)

Section 5.10.3 provides details of the login process as well as the details of changing the identity on the local system.

6.2 Audit

Section 5.6 provides details of the SLES system's 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 *filter.conf* audit configuration file. Section 5.14.4 describes how configuration parameters are loaded into the SLES kernel.

6.2.2 Audit processing (AU.2)

Sections 5.6.1 and 5.6.2 provide details of how processes attach and detach themselves from the audit subsystem. Section 5.14.1 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 5.14.3 describes information stored in each audit record.

6.2.4 Audit post-processing (AU.4)

Section 5.14.2 describes audit subsystem utilities provided for post-processing of audit data.

6.3 Discretionary Access Control

Section 5.1 and 5.2 provide details on Discretionary Access Control on the SLES system.

6.3.1 General DAC policy (DA.1)

Section 5.1 and 5.2.2 provide details on functions that implement general Discretionary Access Control policy.

6.3.2 Permission bits (DA.2)

Section 5.1.1.1, 5.1.2.1 and 5.1.7.1 provide details on calls that perform Discretionary Access Control based on permission bits.

6.3.3 Access Control Lists (DA.3)

Sections 5.1.2.1 and 5.1.7.2 provide details on Discretionary Access Control based on access control lists on file system objects.

6.3.4 Discretionary Access Control: IPC objects (DA.4)

Section 5.3.3 provides details on Discretionary Access Control for IPC objects.

6.4 Object reuse

Sections 5.1, 5.2, 5.3, 5.4 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 5.5.3 provide details on object reuse handling for memory objects.

6.5 Security management

Section 5.12 provides details on various commands used to perform security management.

6.5.1 Roles (SM.1)

Section 5.12 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 file system system calls 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 5.10.3 and 5.12 provide details on various commands used to manage authentication databases.

6.5.4 Management of audit configuration (SM.4)

Sections 5.14.2 and 5.14.4 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 5.11.1, 5.11.2 and 5.11.4 describe secure communications protocols supported by SLES.

6.6.1 Secure protocols (SC.1)

Section 5.11.2 describes the Secure Shell (SSH) protocol. Section 5.11.1 describes the Secure Socket Layer (SSL) protocol. Section 5.11.1.3 describes cipher suites and cryptographic algorithms supported by SLES.

6.7 TSF protection

Section 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 SLES system.

6.7.6 Internal TOE protection mechanisms (TP.6)

Section 4.1.1 describes hardware privilege implementation for the xSeries, pSeries, iSeries, zSeries, and Opteron eServer 325. Section 5.5.1 describes memory management and protection. Section 5.2 describes process control and management.

6.7.7 Testing the TOE protection mechanisms (TP.7)

Section 5.15 describes the 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 & 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 discusses the interfaces between subsystems, but it will only discuss 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 Kernel Subsystem Interfaces: Summary

This section identifies the kernel subsystem interfaces and structures them per kernel subsystem into:

- External Interfaces
  Those are the system calls associated with the subsystem. Those system calls are strcutured into “TSFI System Calls” and “Non-TSFI System Calls”.

- Internal Interfaces
  Those are interfaces not 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 is always the name of the system call prefixed with “sys_” (for example, for a system call “xyz” the kernel internal entry point is “sys_xyz”).

- Data Structures
  Kernel subsystem maintain 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 further described in this chapter. To obtain the information on the purpose of the system call, its parameter, return code, restrictions and effects, the reader is referred to the man page for the system call, which is part of the functional specification. The spreadsheet delivered as part of the functional specification shows also, on which platform the system call is available.

Concerning the internal interfaces, this chapter contains a reference where to find the description of the function implementing this internal interface. This may either be a reference to another chapter of this document or a reference to another document or book, which are part of the high level design of the TOE.

Conncerning 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 (System calls)

1. TSFI System Calls

access
chdir
chmod
chown
creat
execve
fchmod
fchown
fremovexattr
fsetxattr
ioctl
lchown
link
lremovexattr
lsetxattr
mkdir
mknod
mount
open
removexattr
rename
rmdir
setxattr
symlink
truncate

umask
unlink
utime
utimes

2. Non-TSFI System Calls

chroot
close
dup
dup2
epoll_create
epoll_ctl
epoll_wait
fadvise
fchdir
fcntl
fdatasync
fgetxattr
flistxattr
flock
fstat
fstatfs
fsync
ftruncate
getdents
getxattr
io_cancel
io_destroy
io_getevents
io_setup
io_submit
lgetxattr
listxattr
llistxattr
lookup_dcookie
lseek
lstat
mmap
mmap2
msync
munmap
pciconfig_iobase
pciconfig_read
pciconfig_write
pipe
pivot_root
poll
pread
pwrite
quotatcl
read
readahead
readdir
readlink
readv
select

stat
statfs
swapoff
sysfs
umount
ustat
write
writev

6.8.1.1.2 Internal function interfaces — defined in

permission	this document, chapter 5.1.1.1
vfs_permission	this document, chapter 5.1.1.1
get_empty_filp	this document, chapter 5.1.1.1
fget	this document, chapter 5.1.1.1
do_mount	this document, chapter 5.1.1.2

specific ext3 methods:

ext3_create	this document, chapter 5.1.2.1
ext3_lookup	this document, chapter 5.1.2.1
ext3_get_block	this document, chapter 5.1.2.1
ext3_permission	this document, chapter 5.1.2.1
ext3_truncate	this document, chapter 5.1.2.1

specific isofs methods:

isofs_lookup	this document, chapter 5.1.2.2

basic inode operations (create to revalidate are described in [VFS], attribute and extended attribute functions are described in this document in chapter 5.1.2.1 in the context of the ext3 file system):

create
lookup
link
unlink
symlink
mkdir
rmdir
mknod
rename
readlink
followlink
truncate
permission
revalidate
setattr
getattr
setxattr
getxattr
listxattr
removexattr

inode super operations (not to be used by other subsystems, therefore no subsystem interface!)

read_inode2	[ORL], chapter 12
dirty_inode	[ORL], chapter 12
write_inode	[ORL], chapter 12
put_inode	[ORL], chapter 12
delete_inode	[ORL], chapter 12
put_super	[ORL], chapter 12
write_super	[ORL], chapter 12

write_super_lockfs	[ORL], chapter 12
unlockfs	[ORL], chapter 12
statfs	[ORL], chapter 12
remount_fs	[ORL], chapter 12
clear_inode	[ORL], chapter 12
umount_begin	[ORL], chapter 12

dentry operations (not to be used by other subsystems, therefore no subsystem interface!):

d_revalidate	[ORL], chapter 12
d_hash	[ORL], chapter 12
d_compare	[ORL], chapter 12
d_delete	[ORL], chapter 12
d_release	[ORL], chapter 12
d_iput	[ORL], chapter 12

6.8.1.1.3 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)

1. TSFI System Calls

sysctl
capset
clone
fork
ioperm
iopl
kill
modify_ldt
ptrace
reboot
setfsgid
setfsuid
setgid
setgroups
setregid
setresgid
setresuid
setreuid
setuid
swapon
vfork

vm86

2. Non-TSFI System Calls

exit
acct
alarm
arch_prctl
clock_getres
clock_gettime
clock_nanosleep
clock_settime

capget
exit_group
futex
getcwd
getegid
geteuid
getgid
getgroups
getitimer
getpeername
getpgid
getpgrp
getpid
getppid
getpriority
getresgid
getresuid
getrlimit
getrusage
getsid
gettid
gettimeofday
getuid
nice
pause
personality
prctl
restart_syscall
sched_get_priority_max
sched_get_priority_min
sched_getaffinity
sched_getparam
sched_getscheduler
sched_rr_get_interval
sched_setaffinity
sched_setparam
sched_setscheduler
sched_yield
setitimer
setpgid
setpriority
setrlimit
setsid
set_tid_address

stime
sync
sysinfo
tgkill
time
timer_create
timer_delete
timer_getoverrun
timer_gettime
timer_settime
times
tkill
uname
uselib
vhangup
wait4
waitpid

6.8.1.2.2 Internal function interfaces — defined in

current	[ORL], chapter 3
request_irq	[RUBN], chapter 9
free_irq	[RUBN], chapter 9

6.8.1.2.3 Data structures

task_struct	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)

1. TSFI System Calls

ipc (placeholder for all ipc related system calls on x, i, p and z Series)
msgcntl
msgget
msgrcv
msgsnd
semctl
semget
semop
semtimedop
shmat
shmctl
shmget

2. Non_TSFI System Calls

accept
connect
getsockname
getsockopt
listen
recv
recvfrom
recvmsg
rt_sigaction

rt_sigpending
rt_sigprocmask
rt_sigqueueinfo
rt_sigreturn
rt_sigsuspend
rt_sigtimedwait
send
sendfile
sendmsg
sendto
setdomainname
setsockopt
shutdown
sigaction
signal
sigpending
sigprocmask
sigreturn
sigsuspend
socket
socketcall (placeholder for all socket related system calls on x, i, p and z Series)
ssetmask

6.8.1.3.2 Internal function interfaces

	defined in
do_pipe	this document, chapter 5.3.1.1 [ORL], chapter 19
pipe_read	this document, chapter 5.3.1.1 [ORL], chapter 19
pipe_write	this document, chapter 5.3.1.1 [ORL], chapter 19
init_special_inode	this document, chapter 5.3.2.1
fifo_open	this document, chapter 5.3.2.2
ipc_alloc	this document, chapter 5.3.3.2
ipcperms	this document, chapter 5.3.3.2
send_sig_info	this document, chapter 5.3.4.2
unix_create	this document, chapter 5.3.5
inet_create	this document, chapter 5.3.5
sk_alloc	this document, chapter 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)

1. **TSFI System Calls**

bind

2. **Non-TSFI System Calls**

Sethostname
socketpair

6.8.1.4.2 Internal interfaces

Sockets are implemented within the inode structure as specific inode types. `inode.u` in the case of an inode for a socket points to a structure of type `socket`. This structure contains the pointers to the methods for the socket, which are:

- release
- bind
- connect
- socketpair
- accept
- getname
- poll
- ioctl
- listen
- shutdown
- setsockopt
- getsockopt
- sendmsg
- recvmsg
- mmap
- sendpage

The inode is created by the socket system call. The system calls for bind, connect, accept, poll, listen, setsockopt, getsockopt, and ioctl are directly implemented by the methods registered for the socket. read and write as well as send, sendmsg, sendto and recv, recvfrom, recvmsg are implemented by the methods registered for sendmsg and recvmsg, close is implemented by the methods registered for shutdown and release, 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

6.8.1.5 Kernel subsystem memory management

This section lists external interfaces, internal interfaces and data structures of the memory management subsystem.

6.8.1.5.1 External interfaces

1. **TSFI System Calls**

brk

2. **Non-TSFI System Calls**

get_mempolicy
madvice
mbind
mincore

mlock
mockall
mprotect
mremap
munlock
munlockall
rtas
set_mempolicy
swapcontext

6.8.1.5.2 Internal interfaces — defined in

get_zeroed_page	this document, chapter 5.5.2.1 [RUBN], chapter 7
__vmalloc	[RUBN], chapter 7
vfree	[RUBN], chapter 7
kmalloc	[RUBN], chapter 7
kfree	[RUBN], chapter 7
__ get_free_pages	[RUBN], chapter 7
free_pages	[RUBN], chapter 7

6.8.1.5.3 Data structures

mm_struct	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 (System calls)

None

6.8.1.6.2 Internal interfaces

audit_intercept	[LLD], section 7.2
audit_lresult	[LLD], section 7.2
audit_netlink_msg	[LLD], section 7.2

In addition, the audit kernel subsystem has two internal interfaces between itself and the audit device driver:

- the buffer for audit messages
- the kernel structure defining the filter rules

The buffer is filled by the audit kernel subsystem and and emptied by the audit device driver. The filter rules are set by the device driver when it receives them from the audit daemon. The filter rules are used by the kernel subsystem for filter defined selective auditing. The internal structure of the filter rules is defined in the man page of laus_setfilter(3).

6.8.1.6.3 Data structures

audit_filter	laus_setfilter(3) man page

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.

The TOE includes a specific device driver for the audit device. As with most other devices, direct access to this device is restricted to processes running with root privileges. The audit device driver implements a set of ioctl commands a trusted process can use to communicate with the audit device driver. The set of ioctl commands for the audit device driver is defined in the audit(4) man page. A library is provided trusted processes can use to format the parameter for the ioctl system calls to the audit device driver.

6.8.1.7.2 Internal interfaces

Device drivers implement a set of "methods" other kernel subsystems can use directly. In most cases, the File and I/O subsystem will use those methods after its 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.

Except for the audit device drivers security functions as defined in the Security Target are not implemented in the device driver itself. 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 12 describes the methods for block devices.

1. **Character Devices**

Possible Character Device methods are:

- llseek
- read
- write
- readdir
- poll
- ioctl
- mmap
- open
- flush
- release
- fsync
- fasync
- lock
- readv
- writev
- owner

other functions:

- register_chrdev [RUBN], chapter 3
- unregister_chrdev [RUBN], chapter 3

2. **Block Devices**

Possible Block Device Methods are:

- open
- release
- ioctl
- check_media_change
- revalidate

In addition a device specific function `request()`needs to be defined.

other functions:

- register_blkdev [RUBN], chapter 12
- unregister_blkdev [RUBN], chapter 12

6.8.1.7.3 Data structures

device_struct	fs/devices.c
file_operations	include/linux/fs.h
block_device_operations	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)

1. TSFI System Calls

delete_module
init_module

2. Non_TSFI System Calls

nfsservctl
syslog

6.8.1.8.2 Internal interfaces

module dependent

6.8.1.8.3 Data structures

module dependent

6.8.2 Trusted processes interfaces: summary

Trusted processes need to use system calls when they need the functions of a kernel subsystem. The interfaces to the kernel subsystems therefore are the system calls only.

Trusted processes can communicate with each other using the named objects provided by the kernel: 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 behaviour. 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

[ORL] Understanding the LINUX KERNEL, 2nd Edition, Daniel P. Bovet, Marco Cesati, ISBN# 0-596-00213-0

[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.

[STALLINGS] 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

[ISER] IBM iSeries Hardware, ITSO Technical Overview

[ZPOP] z/Architecture Principles of Operation

[TIGR] Linux Kernel 2.4 Internals, Tigran Aivazian

[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.

[YNG] Internet Security Protocols: SSLeay & TLS, Eric Young

[DRKS] *The TLS Protocol version 1*, Tim Dierks, Eric Rescorla

[ENG] PowerPC 64-bit Kernel Internals, Engebretsen David

[BOU] The Linux Kernel on iSeries, David Boutcher

[BIE] Linux Audit-Subsystem Design Documentation for Kernel 2.6, Thomas Biege

[ALM] Booting Linux: History and the Future, 2000 Ottawa Linux Symposium, Almesberger, Werner.

[FEN] 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

[SAMD] SUSE Linux Enterprise Server 8 for AMD64

[KLN] 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

[VA] SLES Vulnerability Assessment, Janak Desai

[LLD] SLES Low Level Design, Janak Desai, George Wilson, Michael Halcrow

[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.

[SCHNEIER] "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.