UNIVERSITY MOULOUD MAMMERI OF TIZI OUZOU
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Operating Systems and System Programming

Level: First year Master cycle
specialty: Intelligent Computer Systems

TABLE OF CONTENTS

List of Figures

List of tables

Abstract

This course offers a comprehensive exploration of operating systems and system programming, with a focus on Linux environments. It covers essential concepts and practical applications across various areas:

The course begins with an in-depth analysis of the Linux boot process, encompassing BIOS/UEFI initialization, kernel loading, and interactions with devices and services. It examines the critical role of the Linux kernel and the management of kernel modules.

Subsequently, the course delves into file systems, elucidating their structures, file types, and manipulation techniques. It provides insights into managing files and directories using command-line tools and explores various file system types utilized in Linux environments.

Another significant aspect covered is file storage mechanisms and sharing protocols. This includes techniques of file storage and access, space management in storage devices, file sharing strategiesm, File locking, File system consistency, inode structure and the role of the memory cache in file access.

Security considerations in Linux environments are thoroughly discussed, covering file and directory permissions, user authentication mechanisms, and firewall configurations. Additionally, students learn about security tools to safeguard against unauthorized access.

Lastly, the course explores network administration under Linux, encompassing network interface configuration, domain name resolution, and the setup of essential network services such as DHCP, DNS, and web servers. Students also gain insights into network monitoring and troubleshooting techniques.

Upon completion of this course, students will have acquired a comprehensive understanding of operating systems principles, file system management, security measures, and network administration within Linux environments.

Chapter I

BOOT PROCESS IN LINUX, KERNEL, MODULES SERVICES AND DEVICES

I.1 Introduction

The boot process in Linux involves several crucial steps, each contributing to the system's initialization and readiness for use. Here's a breakdown of each component:

1. Kernel Booting:
    - When you power on a Linux system, the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) performs hardware initialization tasks.
    - The bootloader, such as GRUB (Grand Unified Bootloader) or LILO (Linux Loader), loads the Linux kernel from the system's storage device (typically a hard drive).
    - The kernel image is usually stored in the /boot directory on the system partition.
2. Kernel Initialization:
    - Once loaded into memory, the Linux kernel initializes and configures essential hardware components, including processors, memory, buses, and devices.
    - The kernel launches the init process, which is responsible for initializing the system further and starting other processes.
3. Module Loading:
    - Linux supports modularization, allowing device drivers and other features to be loaded dynamically as needed.
    - Kernel modules are pieces of code that can be loaded and unloaded into the kernel at runtime. This helps optimize kernel size and memory usage.
    - The modprobe, insmod, and rmmod commands are used to manage kernel modules.
4. Service Startup:

- After kernel initialization, the init process (or systemd in modern distributions) starts various system services.
- Services are programs or daemons that run in the background and provide specific functionality, such as networking, file sharing, printing, etc.
- The init process determines which services should be started based on the system's runlevel or target.

5. Device Detection:
    - During boot, the Linux kernel detects and initializes hardware devices connected to the system, such as hard drives, network interfaces, USB devices, etc.
    - Detected devices are configured and made available to the operating system for use.
    - Device drivers play a crucial role in enabling communication between the kernel and hardware devices.

Overall, the boot process in Linux is a sequence of events that brings the system from a powered-off state to a fully operational state, allowing users to interact with the system and run applications. Each step is essential for the system's proper functioning and performance.

I.2 Boot Process

I.2.1 BIOS

The BIOS (Basic Input Output System) serves as the software interface between hardware and software at a very basic level. It provides the lowest level of interface to device drivers. The BIOS resides on a computer's EEPROM (Electrically Erasable Programmable Read-Only Memory). When the computer is powered on electrically or reset, a signal called PowerGood (or Power_Good, Power_OK, PWR_OK, etc.) is sent to the microprocessor. This triggers the execution of the BIOS. The BIOS performs a power-on self-test and then searches for devices, including those used for booting. Hardware information is permanently stored in a small CMOS memory powered by a battery. At the end of the process, the boot device is selected. The BIOS then reads and executes the first physical sector of the boot media. This is typically the MBR or PBR of the active partition.

I.2.2 Boot Loader

The BIOS activates the Initial Program Loader (IPL) from the first 512 bytes of the boot media. On Linux, the loader is divided into two parts:

- The initial loader of 512 bytes does not contain enough code to provide menus and initiate the loading of an operating system. It loads the second phase, commonly known as the boot loader.
- The second phase provides an interface to launch an operating system from a given choice. The default boot loader on most Linux distributions is called GRUB (Grand Unified Bootloader).

Note that the BIOS is only involved in the machine's startup, the use of the boot loader, and the very early stages of kernel loading. After that, it becomes unnecessary. The kernel has its own detection functions although it relies on BIOS configuration.

I.2.3 Kernel Initialization

During kernel loading, a multitude of information scrolls on the screen. This information cannot be managed at that time. However, just after moving to the next step (init), all kernel traces are placed in the /var/log/dmesg file. The kernel's initial tasks include:

- Detecting and initializing hardware,
- Loading initrd and any modules,
- Mounting the root file system in read-only mode,
- Creating the first console,
- Launching the first process (init).

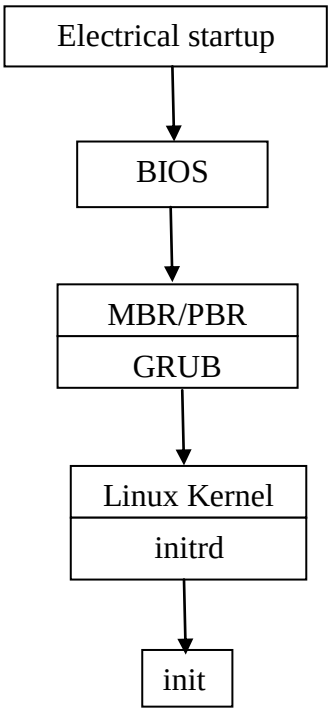


FIG I.1 Linux Boot Sequence

I.3 init

I.3.1 Role of the init program

The init program, the first to start and the last to stop within the system, is responsible for launching all other tasks. Initially, the role of init is to start and stop all services. It is init that executes various initial tasks necessary for the proper functioning of Linux by running several commands and scripts. Once the system is booted and the services are launched, init remains active to manage the state changes of the processes it controls and the runlevels. The init process is the parent of all processes. It always has PID 1. Its configuration is present in the /etc/inittab file. If this file is corrupted and unusable, you will need to boot into single-user mode (S, s, or 1) and repair it, or at worst, boot from an external media or rescue disk. /etc/inittab is a fundamental file of the operating system.

I.3.2 Runlevels

A runlevel is a state in which Linux operates. This state is controlled by init. Each state has its own configuration (either through inittab or through scripts called initscripts). A runlevel can, for example, be used to run Linux in single-user mode, multi-user mode, with or without networking, with or without a graphical interface. All runlevels are customizable by the administrator. On various Linux distributions, these runlevels are generally defined as follows:

Table I.1 Runlevels on various Linux distributions

Runlevel	Effect
0	Halt: Shuts down the operating system, powers off the machine
1	Single-User Mode: Used for maintenance, console mode
2	Multi-User Mode: No networking, console mode
3	Multi-User Mode: With networking, console mode
4	Reserved for the administrator's discretion
5	Multi-User Mode: With networking, graphical X Window environment
6	Reboot: Reboots the machine
S, s	Single-User Mode: The lowest mode used in case of booting issues

However, the Debian distribution does not differentiate between levels 2, 3, 4, and 5, and starts by default at level 2 where everything is launched, including possibly the graphical interface. The default runlevel is set in /etc/inittab on the initdefault line.

id:2:initdefault:

You can replace 2 with another desired runlevel at boot.

Note that Runlevels 7 to 9 are perfectly valid but are not used by default.

I.3.3 The /etc/inittab file

The behavior of the init process and runlevels is defined in the */etc/inittab* file. The syntax of a line is as follows:

Id:[Runlevels]:Action:Command

Table I.2 Field Descriptions of the inittab File

File	Description
Id	Four-character line identifier, purely indicative (except with getty where it indicates the terminal number)
Runlevels	Indicates whether the command should be considered for the requested runlevel, it is a list of runlevels without a separator
Action	Type of action to perform under the circumstances for this line
Command	The command to execute with its parameters and redirections

These are some of the key actions that can be specified in the inittab file to control the behavior of the init process during boot-up and level transitions. Table I.3 describes the main actions. Here is an example of the /etc/inittab file from a Debian Wheezy installation:

```
# / etc / inittab : init (8) configuration .
# $Id : inittab ,v 1.91 2002/01/25 13:35:21 miquels Exp $
# The default runlevel .
id :2: initdefault :
# Boot - time system configuration / initialization script .
# This is run first except when booting in emergency (-b) mode .
si :: sysinit :/ etc / init .d/ rcS
# What to do in single - user mode .
~~: S: wait :/ sbin / sulogin
# / etc / init .d executes the S and K scripts upon change
# of runlevel .
#
# Runlevel 0 is halt .
# Runlevel 1 is single - user .
# Runlevels 2-5 are multi - user .
# Runlevel 6 is reboot .
l0 :0: wait :/ etc / init .d/rc 0
```

l1 :1: wait :/ etc / init .d/rc 1
l2 :2: wait :/ etc / init .d/rc 2
l3 :3: wait :/ etc / init .d/rc 3
l4 :4: wait :/ etc / init .d/rc 4
l5 :5: wait :/ etc / init .d/rc 5
l6 :6: wait :/ etc / init .d/rc 6
 # Normally not reached , but fallthrough in case of emergency .
z6 :6: respawn :/ sbin / sulogin
 # What to do when CTRL -ALT - DEL is pressed .
ca :12345: ctrlaltdel :/ sbin / shutdown -t1 -a -r now
...

Table I.3 Actions that define the activities of init during boot-up and level changing

Action	Description
respawn	Restarts the command if it terminates.
wait	Waits for the command to finish before moving on to the next one.
once	Executes the command only once, then ignores the line.
boot	Executes the command at system boot-up.
bootwait	Executes the command at boot-up and waits for it to finish before moving on to the next one.
initdefault	Sets the default runlevel at boot-up.
sysinit	Executes the command at system boot-up before running runlevel initialization scripts.
ctrlaltdel	Specifies the action to take when the Ctrl+Alt+Del key combination is pressed.
kbrequest	Specifies the action to take when a specific key sequence is pressed on the keyboard.

I.3.4 Changing Runlevels

You can change runlevels dynamically after the machine has booted using the command /sbin/init. The following command switches to runlevel 5:

$ sudo init 5

You can specify the values q (or Q), u (or U), or -t:

- q or Q: init rereads the /etc/inittab file if it has been modified, correcting its internal tables.

- u or U: init restarts without rereading inittab and without changing runlevels. If services have been added or removed from the current runlevel, init takes the modification into account.
- -t: When init has finished shutting down services (or rather when the rc script has done so, see below), init sends the SIGTERM signal to all remaining processes, asking them to terminate properly, waits for the specified number of seconds (5 by default), then sends them SIGKILL. The current runlevel can be seen with the command /sbin/runlevel. The first value returned is the previous runlevel before the current one. An N means there is no previous level. The second value is the current runlevel.

$ sudo runlevel
N 2

I.3.5 Basic System Setup

Regardless of the default runlevel specified, init always executes the command associated with the sysinit, bootwait, or boot actions from /etc/inittab when the system boots; the sysinit action being the first.

si::sysinit:/etc/init.d/rcS

The script calls all scripts from the S (Single) runlevel, and the following tasks are executed in the order listed:

1. Configuring kernel parameters present in /etc/sysctl.conf.
2. Setting up device files.
3. System clock configuration.
4. Loading keyboard character tables.
5. Activating SWAP partitions.
6. Setting the hostname.
7. Controlling and mounting the root file system (this time in read-write mode).
8. Enabling disk quotas.
9. Controlling and mounting other file systems.
10. Cleaning stale locks and PID files in case of a sudden shutdown.

I.3.6 Runlevels

The script /etc/init.d/rc takes as a parameter the default runlevel according to the initdefault line of /etc/inittab or the one specified when manually calling the init command. It initializes the desired runlevel and is responsible for starting and stopping associated services when the runlevel changes.

l0:0:wait:/etc/init.d/rc 0

l1:1:wait:/etc/init.d/rc 1

...

l6:6:wait:/etc/init.d/rc 6

I.3.7 Managing Runlevels and Services

I.3.7.1 Concept of Services

A service (or daemon) is an application that runs in the background, waiting to be used or performing certain critical system tasks. A service is often associated with a server. For example, the Apache web server uses a service named apache2. This service handles requests arriving on the HTTP port of the machine where Apache is installed and then returns the requested web page to the sender.

I.3.7.2 Services in init.d

The runlevel defines the services to be started for that level. It is the rc script that loads the services. Services are controlled (started, stopped, restarted, checked status, etc.) using scripts located in /etc/init.d.

ls -l /etc/init.d
total 420
-rwxr-xr-x 1 root root 2227 Apr 1 2012 acpid
-rwxr-xr-x 1 root root 5195 Sept 6 2011 alsa-utils
-rwxr-xr-x 1 root root 2014 May 21 2012 anacron
-rwxr-xr-x 1 root root 7820 Oct 30 2012 apache2
...

For each runlevel N, there is a directory /etc/rcN.d that contains symbolic links to the services present in /etc/init.d to start or stop. The prefix of each link's name defines its startup or shutdown order. The name follows this format: [SK]NNservice

- S: start.
- K: kill (stop).
- NN: numerical order of startup or shutdown. (00=first, 99=last).
- service: service name.

For example, the link S12networking in the directory /etc/rcS.d indicates that the networking service, responsible for setting up the network, will be started with order 12; after S01, S02, ..., S11, and before S13, S14, ..., S19.

ls -1 /etc/rcS.d
S01hostname.sh

S01mountkernfs.sh
...
S12networking
...
S19x11-common

When rc is executed, it first lists all links starting with K* using a for loop. Then it does the same for S* but this time, it starts the services.

I.3.7.3 Manual Control of Services

Via the script

Services can be manually started. Each service in /etc/init.d accepts at least two parameters:

- start: starts the service.
- stop: stops the service.

For example, if you want to start and stop the ssh service manually:

$ sudo / etc / init .d/ ssh start
Starting OpenBSD Secure Shell server : sshd .
$ sudo / etc / init .d/ ssh stop
Stopping OpenBSD Secure Shell server : sshd .

Some services may accept additional parameters:

$ sudo /etc/init.d/ssh
[info] Usage: /etc/init.d/ssh {start | stop | reload | force-reload | restart | try-restart | status}.

status: Provides the status of the service (started or not). Depending on the service, additional information may be provided.

reload/force-reload: Instructs the service to reload its configuration.

restart: Stops and restarts the service, regardless of the outcome of the stop.

try-restart: Stops and restarts the service only if the stop was successful.

Via The service command

The **service** command (**/usr/sbin/service**) simply allows you to avoid specifying the path to the service startup script and instead use just the service name:

$ sudo service ssh start
Starting OpenBSD Secure Shell server : sshd .
$ sudo service ssh stop
Stopping OpenBSD Secure Shell server : sshd .

I.3.7.4 Modifying Runlevels

The update-rc.d command creates the necessary links in various rcN.d directories. Here are two examples of syntax. The first one registers a service with default parameters. In this case, the service is configured to start at runlevels 2 to 5 and stop at runlevels 0, 1, and 6. The start/stop position is at 20.

$ sudo update-rc.d ssh defaults

In the second example, the service is inserted with complete options. Upon startup, the service is at position 10 in runlevels 3, 4, and 5. Upon shutdown, the service is at position 5 in runlevels 0, 1, and 6. Don't forget the dots.

$ sudo update-rc.d ssh start 10 3 4 5 . stop 05 0 1 6 .

The **remove** parameter removes the links from various directories. However, the associated script **/etc/init.d/xxx** must no longer exist itself. Otherwise, use the **-f** parameter to force the removal of links (the script itself remains in place).

$ sudo update-rc.d -f ssh remove

I.3.8 Virtual Consoles

Virtual consoles provide virtual terminals on a machine. They are defined in **/etc/inittab** and are available via the **/dev/ttyN** devices where N is the console number.

- Switching between consoles can be done with the key sequence [Alt][Fn] (for example, [Alt][F2]) from the console or [Ctrl][Alt][Fn] from X Window.
- Use [Alt][Right Arrow] and [Alt][Left Arrow] to switch to the next or previous console.
- **/dev/ttyN** represents virtual console N.
- **/dev/tty0** represents the current console.
- By default, there can be 12 virtual terminals due to the 12 function keys, but only 6 are enabled by default.
- X Window is launched by default on the first available console, usually the 7th.

Consoles are launched by **inittab** and **getty** processes. These are the only **inittab** entries where the label is important: it corresponds to the console number.

I.3.9 Logins

Once the terminals (getty) are launched by init, a prompt awaits the user's input for their username (login). Before this prompt, the contents of the **/etc/issue** file are displayed. Upon entering the username, the terminal executes the **login** command (**/bin/login**), which

prompts for the password. If the password is correct (verified in **/etc/passwd** and **/etc/shadow**), then **login** displays the contents of **/etc/motd** and executes a shell (always defined in **/etc/passwd**).

Note that **getty** and **login** do not fork: the launched processes are not child processes but instead replace the current process (using the exec API). There are no parent-child relationships between processes (**getty -> login -> shell**), but each one replaces the previous, and the process retains the same PID. This is how init knows when a connection is terminated. Once the session is ended (shell exit), init restarts a terminal for a new connection (respawn command).

I.3.10 System Shutdown

Several methods exist to properly shut down a machine running Linux. Firstly, shutdowns are managed by init with runlevels 0 and 6. In practice, both are almost identical except for the final action:

- Runlevel 0: The computer is powered off.
- Runlevel 6: The computer reboots.

Thus, the following command shuts down the computer:

$ sudo init 0

and this one reboots it:

$ sudo init 6

However, the most correct, clean, and secure command to shut down the system is **shutdown**, which calls init but accepts additional parameters. Its basic syntax is:

shutdown <option> <time> <message>

The parameters are given in Table I.4.

These parameters can be used with the **shutdown** command to control the shutdown process of the system.

The delay can be specified in different ways:

- **hh:mm**: at a specific time.
- **+m**: in m minutes.
- **now**: an alias for +0, meaning immediately.

The following command schedules a reboot in 15 minutes with a warning message:

$ sudo shutdown -r +15 "Reboot for maintenance in 15 minutes"

Table I.4 Parameters for the shutdown command:

Parameter	Description
-h	Halt the system after shutdown
-r	Reboot the system after shutdown
-c	Cancel a scheduled shutdown
-k	Do not actually shutdown, just send warning
-t	Time before shutdown, in minutes
-f	Close all user sessions
-W	Warn users before shutdown
-wall	Send a message to all users

This will display a broadcast message indicating the reboot for maintenance in 15 minutes. The following command cancels the reboot.

$ sudo shutdown -c "Maintenance canceled"
Broadcast message from root@debian32 (pts/1) (Sat Jan 25 23:15:33 2014):
Maintenance canceled

The reboot and halt commands are invoked at the end of runlevels 6 and 0, respectively. If they are called in a runlevel other than 6 or 0, they are equivalent to calling shutdown:

- halt: shutdown -h
- reboot: shutdown -r

I.4 Viewing System Logs

I.4.1 The dmesg Command

The dmesg command retrieves kernel messages generated during system startup and afterwards. The dmesg buffer is circular, meaning that after a certain number of messages, the earliest ones will be overwritten. Administrators or operators typically use this command to check for any potential errors, as messages continue to arrive after boot, especially during hotplug device connection, module loading, crashes, or filesystem corruption. The following example displays only a few lines; the complete output contains 838 lines. It shows what happens when a USB key is inserted during normal usage (after boot).

$ dmesg
...

[510.340026] usb 1-1: new high-speed USB device number 3 using ehci_hcd
[510.474366] usb 1-1: New USB device found, idVendor=058f, idProduct=6387
[510.474372] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[510.474377] usb 1-1: Product: Mass Storage
[510.474380] usb 1-1: Manufacturer: Generic
[510.474384] usb 1-1: SerialNumber: BFF732CC
[510.474977] scsi3: usb-storage 1-1:1.0
...
[511.820253] sdb: sdb1 sdb2 < sdb5 >
...

To analyze the result of dmesg, it's best to either redirect it to a file for offline analysis or use the grep command when searching for specific information.

$ dmesg | grep CPU
[0.000000] SMP: Allowing 8 CPUs, 6 hotplug CPUs
...
[0.143067] CPU0: Intel Pentium(R) Dual-Core CPU E5800 @ 3.20 GHz ...
...

I.4.2 The /var/log/messages File

/var/log/messages is the central file for system messages, whether from the kernel or services. Its content reflects the overall state of the system (not just the kernel) during its usage. The file can contain thousands of lines, especially if a problem occurs.

$ sudo wc -l /var/log/messages
31556 /var/log/messages

Just like with the dmesg command, grep (or tail, head, etc.) is usually used to select the desired lines.

$ sudo grep -i "wifi" /var/log/messages
Feb 16 08:38:37 localhost kernel: [10.477954] iwl3945 0000:0c:00.0: Detected Intel Wireless WiFi Link 3945 ABG

I.5 Services and Kernel Modules

I.5.1 Overview

The kernel is the core of the Linux operating system. Linux itself is just the name of the kernel originally developed by Linus Torvalds. The Linux operating system consists of the kernel and basic operating tools. The Linux kernel resides in the /boot directory, and its name conventionally starts with *vmlinuz.*

*$ ls /boot/vmlinuz**

/boot/vmlinuz-3.2.0-4-686-pae

I.5.2 uname

The uname command (Unix name) provides detailed and comprehensive information about the Linux kernel version. It can take several parameters, including those described in Table I.5.

$ uname
Linux
$ uname -m
i686
$ uname -n
debian
$ uname -r
3.2.0-4-686-pae
$ uname -o
GNU/Linux
$ uname -v
#1 SMP Debian 3.2.53-2
$ uname -a
Linux debian 3.2.0-4-686-pae #1 SMP Debian 3.2.53-2 i686 GNU/Lin

Table I.5 uname Command Parameters and Descriptions

Parameter	Description
-s	Display the kernel name.
-n	Display the network node hostname.
-r	Display the kernel release.
-v	Display the kernel version.
-m	Display the machine hardware name.
-p	Display the processor type.
-i	Display the hardware platform.
-o	Display the operating system.
-a	Display all available information.

I.5.3 Module Management

The core components (scheduler, memory management, process management, APIs, etc.) are still present within a single program (the vmlinuz file). But some device drivers, file systems, extensions, network protocols, etc. may be present in the form of modules. Modules communicate with the kernel via a common API. They run in kernel space. They are

configurable. They can be loaded and unloaded on demand, thus avoiding a machine restart. Modules are located in the directory /lib/modules/$(uname -r).

Modules have a name ending in .ko (kernel object). These are dynamically linked object files to the kernel providing additional APIs.

$ file /lib/modules/$(uname -r)/kernel/fs/ntfs/ntfs.ko
/lib/modules/3.2.0-4-686-pae/kernel/fs/ntfs/ntfs.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
BuildID[sha1]=0a81535be2f8e27137b643777d6d07caf3171da2, not stripped

The word 'relocatable' indicates that it is a relocatable object file.

I.5.3.1 lsmod

The lsmod command lists the currently loaded modules, along with their potential dependencies.

$ lsmod

Module	*Size*	*Used by*	
nls_utf8	*12416*	*1*	
nls_cp	*437 12417*	*1*	
vfat	*17116*	*1*	
fat	*44497*	*1*	*vfat*

...

The first column indicates the name of the loaded module. Its name often reflects its purpose. The second column provides the size of the module. The third column provides a usage counter (how many system components are accessing the module). The last column provides a list of modules using, and thus depending on, the first one.

In the previous example, the module "fat" is used by the module "vfat". In fact, lsmod simply formats the contents of the virtual file /proc/modules.

$ cat /proc/modules
nls_utf8 12416 1 - Live 0xf8990000
nls_cp437 12417 1 - Live 0xf8949000
vfat 17116 1 - Live 0xf899c000
fat 44497 1 vfat, Live 0xf8b28000
...

I.5.3.2 depmod

The depmod command updates the dependency tree between modules by modifying the modules.dep file. The file /lib/modules/$(uname -r)/modules.dep contains two columns. The first one is the path of the module, and the second one is the list of dependencies: the modules that must also be loaded for the first one to function. Here's an example of the line corresponding to the "vfat" module:

$ grep vfat /lib/modules/$(uname -r)/modules.dep

kernel/fs/fat/vfat.ko: kernel/fs/fat/fat.ko

The "vfat.ko" module depends on the "fat.ko" module. Therefore, the "fat.ko" module must be loaded first for "vfat.ko" to function.

The most common usage of depmod is with the -a parameter, which rebuilds the dependencies for all modules corresponding to the current kernel. This action is performed every time the system boots, but if new modules are installed, you must manually run this command to account for the new dependencies.

$ sudo depmod -a

I.5.3.3 modinfo

The *modinfo* command provides all necessary information about a module:

- The filename corresponding to the file,
- A description of the module,
- Its author,
- Its license,
- Its dependencies,
- ...

$ sudo modinfo vfat

filename: /lib/modules/3.2.0-4-686-pae/kernel/fs/fat/vfat.ko

author: Gordon Chaffee

description: VFAT filesystem support

license: GPL

depends: fat

...

I.5.3.4 insmod

The insmod command loads a given module without managing dependencies. It takes a module name with its optional path as a parameter. It is up to the administrator to manage the loading order of modules to avoid errors related to unresolved symbols caused by dependency issues. In the following example, since the modules "fat" and "vfat" are not loaded, an attempt is made to load the "vfat.ko" module alone. An error occurs because this module depends on the presence of "fat.ko".

$ lsmod | grep fat

$ sudo insmod /lib/modules/$(uname -r)/kernel/fs/fat/vfat.ko

Error: could not insert module /lib/modules/3.2.0-4-686-pae/kernel/fs/fat/vfat.ko: Unknown symbol in module

In the next example, the "fat" module is first loaded, then the "vfat" module. There is no error because all dependencies are satisfied.

$ sudo insmod /lib/modules/$(uname -r)/kernel/fs/fat/fat.ko

$ sudo insmod /lib/modules/$(uname -r)/kernel/fs/fat/vfat.ko

I.5.3.5 rmmod

The rmmod command unloads the provided module argument. It is the opposite of insmod, and like insmod, rmmod does not manage dependencies. The administrator should know:

- It is not possible to unload a module in use.
- It is not possible to unload a module if it is used by another module, even if the latter is not used (dependency problem).

In the following example, a USB key containing a vfat filesystem is plugged in and mounted. The first attempt to remove "vfat" fails.

$ mount | grep vfat

/dev/sdb1 on /mnt type vfat (rw, ...)

$ sudo rmmod vfat

Error: Module vfat is in use

In the second example, the key is unplugged. The "fat" and "vfat" modules are now unnecessary. An attempt to remove the "fat" module returns an error related to dependencies.

$ sudo umount /mnt

$ sudo rmmod fat

Error: Module fat is in use by: vfat

In this last example, the "vfat" module is unloaded first, then the "fat" module, in that order, and there are no errors.

$ sudo rmmod vfat

$ sudo rmmod fat

I.5.3.6 modprobe

The modprobe command loads the given module along with all its dependencies. The -r parameter allows unloading a module and those depending on it (if they are not in use). Loading the "vfat" module using modprobe will automatically load the "fat" module.

$ sudo modprobe vfat
$ lsmod | grep fat
vfat 17116 0
fat 44497 1 vfat

Now, seeing that only "vfat" uses "fat" (count is 1), and nothing uses "vfat" (count is 0), you can attempt to unload "vfat" and the modules it depends on if they are no longer used.

$ sudo modprobe -r vfat

I.5.4 Module Loading at Boot

I.5.4.1 initrd

Some modules may be necessary during the boot process, especially for mounting a filesystem. How to mount the root partition in ext4 when the module managing this type of filesystem is on this partition (and not in the kernel)? These modules are placed in an initial ramdisk image or initrd (initial ramdisk). These compressed files are loaded at boot.

*$ ls /boot/initrd**
/boot/initrd.img-3.2.0-4-686-pae

I.5.4.2 The /etc/modules File

In Debian, you just need to add the names of the modules to be loaded at boot to the /etc/modules file. For each line, a modprobe is executed.

$ cat /etc/modules

...

loop

I.5.5 Dynamic Parameters

I.5.5.1 /proc and /sys

/proc and /sys are virtual filesystems containing information about the running kernel. Virtual filesystems take up no space in memory or on any disk. Don't be misled by the size of the pseudo-files contained within them. Almost all files (or directories) contained in /proc and /sys can be read and displayed directly.

The files in /proc provide a wealth of information about the system:

- cpuinfo: Details about the processor(s).
- devices: Present devices.
- meminfo: Global memory status.
- loadavg: System load.
- version: Linux version details.

- modules: Equivalent to the result of lsmod.
- swaps: List and status of swap partitions.
- partitions: List and status of known partitions in the system.
- ...

$ cat /proc/cpuinfo
processor: 0
vendor_id: GenuineIntel
cpu family: 6
model: 23
model name: Pentium(R) Dual-Core CPU E5800 @ 3.20GHz
...

$ cat /proc/version
Linux version 3.2.0-4-686-pae (debian-kernel@lists.debian.org)
(gcc version 4.6.3 (Debian 4.6.3-14)) #1 SMP Debian 3.2.53-2

/proc contains subdirectories that group information by theme:

- /proc/bus: Bus information.
- /proc/net: Network information.
- /proc/sys: Kernel parameters and dynamic configuration.
- /proc/<PID>: Process PID information.
- ...

Some entries in the /proc/sys and /sys filesystems are different from others because their content can be modified, and the modifications are directly taken into account by the kernel without needing to restart the machine.

As an example, here's how to change the number of file handles from 614875 to 205483:

$ sudo echo "205483" > /proc/sys/fs/file-max

I.5.5.2 sysctl

Kernel parameters can be modified during runtime using the sysctl command. The modified values are not saved. In case of a restart, the parameters need to be set again. To make the values permanent (reset at each boot), you need to modify the /etc/sysctl.conf file and then execute the command:

$ sudo sysctl -p

The sysctl -a command allows displaying the values of all parameters:

$ sudo sysctl -a

...

net.ipv4.tcp_timestamps = 1

net.ipv4.tcp_window_scaling = 1

net.ipv4.tcp_sack = 1

...

This command displays the current values of various network-related parameters, such as TCP timestamps, window scaling, and selective acknowledgments (SACK).

I.6 Device Files

I.6.1 Introduction

Consider the operation of devices, which is generally the same on all computers. Devices are connected to a controller, for example, an IDE or SATA controller for IDE disks, a SCSI controller for disks, drives, and other SCSI scanners, or a USB controller. A controller can typically control several devices attached to it. The controller communicates with the microprocessor and memory using buses (command and data buses). On Linux systems, the controller and its devices are managed using drivers (one driver for the controller and one or more drivers for the devices attached to it, for example, a driver for the SCSI controller, then a driver for disks, another for scanners, and yet another for a CD-ROM). The driver is often a kernel add-on module, supplied by the manufacturer or already present. Devices are seen as files. Therefore, processes access devices through these files using C language primitives whose code is in the kernel. The process must first open the special file of the device (open primitive), then read (read) or write (write) data to or from the device as it would with a normal file. These read/write operations are then interpreted by the device driver.

The necessary information for addressing the relevant driver for the device opened by a process is found in the special file **/dev/<device>**.

I.6.2 Special Files

Special device files are conventionally placed in the **/dev** directory and, like any other file, have a unique inode. Therefore, you can know their attributes using the **ls -l** command.

$ ls -l /dev

...

crw------- 1 root root 10, 1 Jan 25 18:09 psaux

...

brw-rw----T 1 root disk 8, 1 Jan 25 18:09 sda1

...

brw-rw----T+ 1 root cdrom 11, 0 Jan 25 18:09 sr0

...

The first character identifies the type of device:

- c: Character device.
- b: Block device.

These modes differentiate the type of data exchange between the filesystem module and the device driver. In character mode, there is no use of system buffers, and the exchange is byte by byte. In block mode, the system accesses the device via an index representing the coordinates of the data block on the media. It is therefore faster for devices like disks. The other two essential attributes of a device file are the pair of information found in place of the file size: the major number and the minor number.

- The major number identifies the driver and hence the device controller.
- The minor number generally identifies the device, but it can also designate a peculiarity of the device, such as the partition of a disk, a specific location, the card number (in the presence of multiple identical controller cards, multiple sound cards, etc.).

Here are some common special files:

- /dev/mem: Physical memory.
- /dev/console: Master console.
- /dev/tty: Standard input/output of the current process.
- /dev/null: Linux's "null device". You can write to it. Reading from it results in an EOF.
- /dev/root: Special root filesystem.
- /dev/ttyS0: COM1 serial port.
- /dev/bus/usb/*: USB devices.
- /dev/sdX: SATA or SCSI disks.
- ...

I.6.3 Knowing Your Hardware

I.6.3.1 PCI Bus

The lspci command provides detailed information about the cards and adapters connected to the PCI bus. Adapters can be those connected to the expansion ports of the motherboard as well as those integrated into the motherboard itself (IDE/SATA controllers, network cards, etc.). AGP and PCI Express buses are considered PCI buses as well.

$ lspci
...
01:00.0 VGA compatible controller: NVIDIA Corporation GF108 [GeForce GT 630] (rev a1)
...
02:00.0 Ethernet controller: Broadcom Corporation NetLink BCM57780 Gigabit Ethernet PCIe (rev 01)
03:02.0 Ethernet controller: Sundance Technology Inc / IC Plus Corp IC Plus IP100A Integrated 10/100 Ethernet MAC + PHY (rev 31)

You can get even more details with the -v option and specify an adapter with its identifiers. To obtain detailed information about the Ethernet controller (02:00.0), you proceed like this:

$ lspci -v -s 02:00.0
02:00.0 Ethernet controller: Broadcom Corporation NetLink BCM57780 Gigabit Ethernet PCIe (rev 01)
Subsystem: Dell Device 0228
Flags: bus master, fast devsel, latency 64, IRQ 17
Memory at fe5fe000 (32-bit, non-prefetchable) [size=8K]
Capabilities: <access denied>
Kernel driver in use: b44

I.6.3.2 USB Bus

The lsusb command does the same thing as lspci but for the USB bus:

$ lsusb
Bus 001 Device 007: ID 125f:c96a A-DATA Technology Co., Ltd. C906 Flash Drive
Bus 001 Device 002: ID 0480:a007 Toshiba America Info. Systems, Inc. External Disk USB 3.0
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
...

When possible, Linux indicates the names of the devices via an identification database, similar to PCI cards. These identifiers also help Linux determine which USB driver to load. As with lspci, you get more information with -v, as well as -d:

$ lsusb -d 125f:c96a

Bus 001 Device 007: ID 125f:c96a A-DATA Technology Co., Ltd. C906 Flash Drive

$ lsusb -v -d 058f:6387

Bus 001 Device 007: ID 125f:c96a A-DATA Technology Co., Ltd. C906 Flash Drive

Device Descriptor:

bLength 18

bDescriptorType 1

bcdUSB 2.00

bDeviceClass 0 (Defined at Interface level)

bDeviceSubClass 0

bDeviceProtocol 0

bMaxPacketSize0 64

idVendor 0x125f A-DATA Technology Co., Ltd.

I.6.3.3 Hardware Resources

The virtual filesystem /proc contains abundant information about hardware. The following table lists some files, specifying the nature of the information they contain:
To access this information, you simply view the contents of the corresponding virtual file using commands such as cat, less, more, etc.

Table I.6 Hardware resources

Virtual File	Information Contained
/proc/interrupts	Interrupts
/proc/dma	DMA Channels
/proc/ioports	I/O Port Address Ranges
/proc/devices	Devices (block and character)
/proc/partitions	Partitions
/proc/cpuinfo	Processor

I.6.3.4 Other Tools

hwinfo

The hwinfo command (provided by the package of the same name, not installed by default) detects hardware and provides a list of it (in a concise manner with the --short option). It relies on querying the hardware for information:

$ sudo hwinfo --short
cpu:
Pentium(R) Dual-Core CPU E5800 @ 3.20GHz, 1200 MHz
Pentium(R) Dual-Core CPU E5800 @ 3.20GHz, 1200 MHz
keyboard:
/dev/input/event1 USB Keykoard
mouse:
/dev/input/mice Colorado USB Optical Mouse
monitor:
Generic Monitor

graphics card:
nVidia VGA compatible controller
sound:
Intel 82801G (ICH7 Family) High Definition Audio Controller
nVidia Audio device
storage:
Intel 82801G (ICH7 Family) IDE Controller
Intel 82801GB/GR/GH (ICH7 Family) SATA IDE Controller
network:
eth0 Broadcom Ethernet controller
eth1 Sundance Technology Inc / IC Plus IC Plus IP100A Integrated 10/100 Ethernet MAC + PHY
network interface:
lo Loopback network interface
eth1 Ethernet network interface
eth0 Ethernet network interface
disk:
/dev/sda TOSHIBA DT01ACA0
/dev/sdb ST3500413AS
/dev/sdc TOSHIBA External USB 3.0
/dev/sdd A USB Flash Drive
...

This command provides a comprehensive overview of various hardware components detected by the system. To obtain detailed hardware information, you can use hwinfo and dmidecode.

hwinfo

hwinfo provides comprehensive hardware details. To specify which component you want to detail, you can use options like --cpu for the processor and --memory for memory. Here's how you can use it:

$ sudo hwinfo --cpu
01: None 00.0: 10103 CPU
[Created at cpu .304]
Unique ID: rdCR . j8NaKXDZtZ6
Hardware Class : cpu
Arch : Intel
Vendor : " GenuineIntel "
Model : 6.23.10 " Pentium (R) Dual - Core CPU E5800 @ 3.20 GHz "
...
Clock : 800 MHz
BogoMips : 3990.28
Cache : 2048 kb
Units / Processor : 2
...

$ sudo hwinfo -- memory
01: None 00.0: 10102 Main Memory
[Created at memory .61]
Unique ID: rdCR . CxwsZFjVASF
Hardware Class : memory
Model : " Main Memory "
Memory Range : 0 x00000000 -0 x178018fff (rw)
Memory Size : 6 GB
Config Status : cfg=new , avail =yes , need =no , active = unknown

dmidecode

dmidecode interprets the Desktop Management Interface (DMI) table of the computer, providing information about the hardware's current state and possible extensions. Unlike hwinfo, which queries specific components like the CPU, dmidecode reads information as detected by the BIOS and motherboard. While it's generally faster and sometimes more accurate than hwinfo, it may provide incorrect information that should be verified.

You can specify which information you want to obtain with -s or -t:

$ sudo dmidecode -t processor
 # dmidecode 2.12
SMBIOS 2.5 present .
Handle 0x0400 , DMI type 4, 40 bytes
Processor Information
Socket Designation : CPU
Type : Central Processor
Family : Core 2 Duo
Manufacturer : Intel
ID: 7A 06 01 00 FF FB EB BF
Signature : Type 0, Family 6, Model 23, Stepping 10
...
$ sudo dmidecode -s processor - frequency
3200 MHz

These commands provide detailed hardware information, allowing you to understand the components of your system better.

Chapter II

Exploring FILE SYSTEMS: STRUCTURE AND MANIPULATION

II.1 Introduction

A file refers to a set of information stored on the disk. The file system is the part of the operating system responsible for managing files. Management involves creation (identification, allocation of disk space), deletion, read and write access, file sharing, and protecting files by controlling access.

II.1.1 What is a file?

For the operating system, a file is a sequence of bytes. However, users may attribute different meanings to the contents of a file (sequences of bytes, sequence of records, tree structure, etc.). Each file is identified by a name to which a location on the disk (a reference) is associated, and it has a set of properties: its attributes. The name generally consists of two parts separated by a dot. The part following the dot is called an extension (e.g., prog.c, header.h, file.doc, archivo.pdf, etc.). The extension can be fixed in size, as in MS-DOS, or variable, as in Unix/Linux; it may be required or not. The extension is necessary in some cases. For example, the C compiler will reject the file prog.txt even if its content is a C program. The name of a file can be case-sensitive: thus, in Unix/Linux, Archivo is not the same as archivo.

II.1.2 File Lifecycle

Files, like many other components, have a lifecycle. They are created (or opened), modified (written to), read from, and ultimately, they may be deleted. This is illustrated in Figure I.1, with system calls that we will study in this chapter.

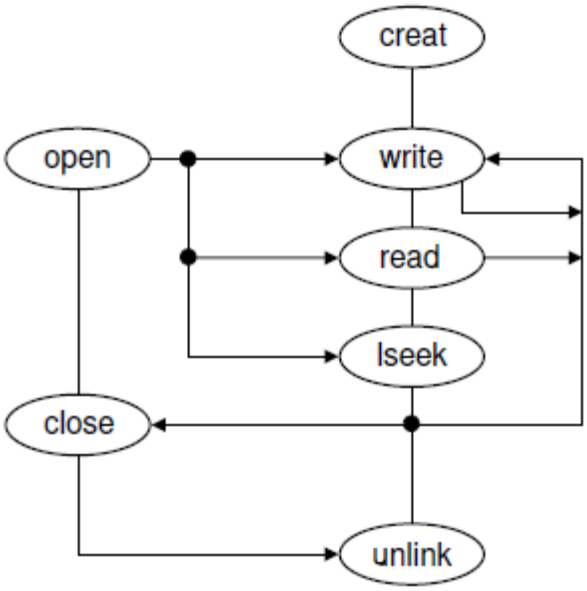


FIG II.1 The lifecycle of a file.

II.1.3 Types of Files

In a system, there are several types of files. Unix/Linux and MS-DOS have regular files, directories, character special files, and block special files. Regular files contain user information and are generally ASCII or binary files. Directories are system files that maintain the structure of the file system. Character special files are related to Input/Output and allow modeling of serial I/O devices such as terminals, printers, and networks. Block special files model disks. In the Unix/Linux system, "-" denotes regular files, "d" denotes directories, "c" denotes character devices, "b" denotes block devices, and "p" denotes named pipes. Devices are files referred to by references. Block and character special files identify physical hardware devices: disks, magnetic tapes, terminals, networks, etc. Each type of device has a controller responsible for its communication. In the operating system, there is a table that points to the various device controllers. All devices are then treated as files. In Unix/Linux, devices such as terminals are special files located under the directory /dev. For example, text typed on the keyboard, like "abcd," can be copied into a file using the command:

leibnitz> cat > file
abcd
^D

II.2 File Access

To access a file, it is necessary to provide the file system with the information needed to locate it on the disk, i.e., to provide it with a path of access. Modern systems allow users to access data within a file directly, without traversing the entire path from the beginning.

II.2.1 File Attributes

File attributes vary from one system to another. However, they can be grouped into two categories:

1. Attributes that serve to control access, such as protection code, password, owner, etc.
2. Attributes that define the type and current state of the file: indicator of ASCII/binary type, current size, creation date, date of last modification, etc.

II.2.2 Inodes

In the Unix/Linux system, all file information is gathered in a structure associated with the file, called an information node, inode, or i-node. The inode contains the following information:

- The type of file (regular, directory, character, block, or pipe).
- A 9-bit protection code.
- Owner's identifier and group.
- Creation date, last access date, and last modification date.
- Reference counter.
- Size, and finally,
- The index table composed of 13 block numbers and pointers. Figure 11.2 shows the typical structure of an inode.

II.3 POSIX File Services

Files allow storing information and retrieving it later. Systems provide a set of file-related system calls. In the case of Unix/Linux, the main POSIX file system calls are:

- open() and creat() for opening a file.
- close() for closing a file.
- read() for reading from a file.
- write() for writing to a file.
- lseek() for moving the file pointer.
- stat() for retrieving information about a file.
- link() for creating a link between two files.
- unlink() for removing a link or a file.

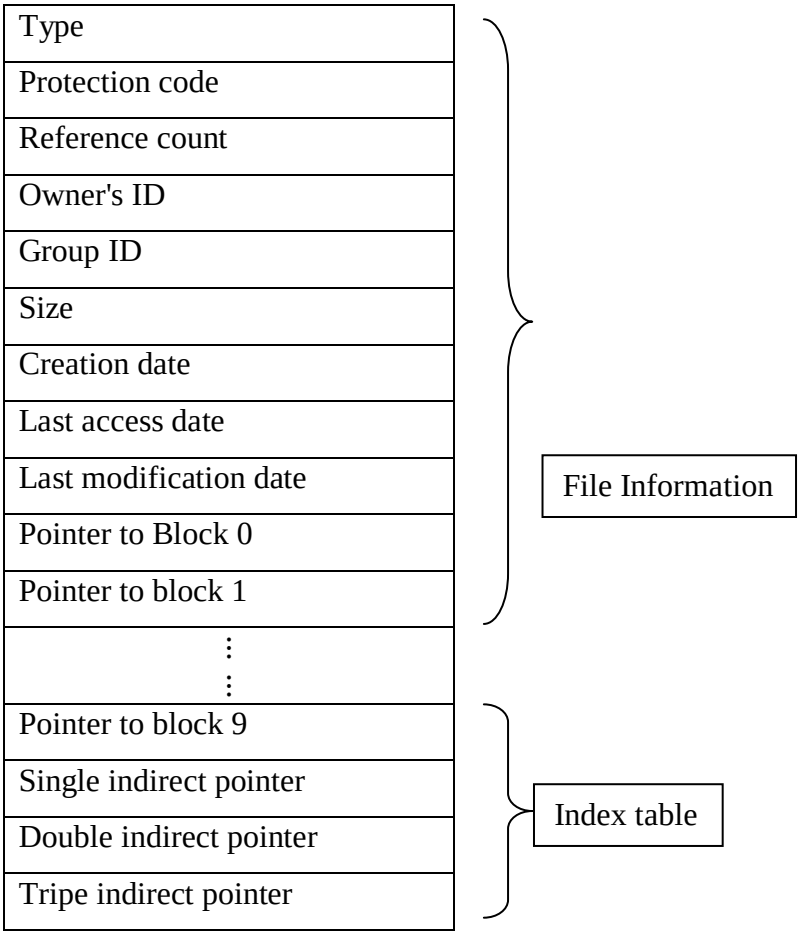


FIG II.2 Inode structure

Opening a file

#include <unistd.h>

#include <fcntl.h>

*int open(char *filename, int mode);*

*int open(char *filename, int mode, int permission);*

*int creat(char *filename, int permission);*

In case of success, open() returns a file descriptor. In case of failure, it returns -1. Each opened file has a pointer that points to an element of the file. This pointer is used to traverse the file. Upon opening the file, the file pointer points to the first element of the file, unless the O_APPEND option has been specified. Reads and writes are done from the current position of the pointer. Each read and each write result in the modification of the pointer's position. The creat() system call is becoming less common. Its usage is equivalent to: open(char *filename, O_WRONLY|O_CREAT|O_TRUNC, int mode) The mode is a

combination of several elements joined by a logical OR. It is mandatory to use one of three constants:

- O_RDONLY: File opened in exclusive read mode.
- O_WRONLY: File opened in exclusive write mode.
- O_RDWR: File opened in read and write mode. Other constants can be used to specify usage more precisely:
- O_CREAT: Create the opened file even in exclusive read mode.
- O_EXCL: Used in conjunction with O_CREAT, this constant ensures that an existing file will not be overwritten. The opening fails if the file already exists.
- O_TRUNC: Used with O_WRONLY or with O_RDWR, it sets the size of an existing file to zero.

Permissions are used when creating a file. They indicate the access permissions for the newly created file. They can be provided directly in octal representation (preceded by a 0) or symbolic constants from the most common set of permissions listed in Table 11.1 can be used. Thus, if we use the combination: "S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH" = 0644 = rw-r--r--, it grants read permissions to all users, and read and write permissions only to the owner of the file.

Examples:

1. This system call creates and then opens a file named "archivo" for read/write access. The file descriptor is fd. The file's protection code is 0600, which corresponds to (rw-------).

```
int fd;
fd = open("archivo", O_CREAT | O_RDWR, 0600);
```

Table II.1 Symbolic Permission Constants

Symbolic constant	Octal value	Meaning of the authorization
S_IRUSR	00400	Read for the owner
S_IWUSR	00040	Write for the owner
S_IXUSR	00100	Execute for the owner
S_IRWXU	00700	Read + write + execute for the owner
S_IRGRP	00020	Read for the group
S_IROTH	00004	Read for everyone
S_IWOTH	00002	Write for everyone
S_IXOTH	00001	Execute for everyone

2. This system call opens the file "archivo" in write mode and positions the pointer at the end of the file. Subsequent writes to the file will therefore append data to the end of it.

```
int fd;
fd = open("archivo", O_WRONLY | O_APPEND);
```

3. The following calls are equivalent:

```
fd = creat("datos.txt", 0751);
fd = open("datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0751);
```

Closing Files:

The close() system call frees a file descriptor fd. The file pointed to by the released descriptor is closed if there are no more descriptors associated with the file.

```
int close(int fd);
```

Reading from a File:

```
#include <unistd.h>
int read(int fd, char* buf, int count);
```

The read() system call reads up to count bytes from the current position of the pointer and copies them into the buffer buf. On success, it returns the number of bytes actually read. The read() system call returns 0 if the end of the file is reached. It returns -1 in case of an error.

II.3.4 Writing to a File:

```
#include <unistd.h>
int write(int fd, void* buf, int count);
```

The write() system call copies count bytes from the buffer buf to the file descriptor fd, starting from the position pointed to by the file pointer. On success, it returns the number of bytes actually written. Otherwise, it returns -1. It should be noted that read() and write() operate on bytes, and it is the programmer's responsibility to interpret them in one form or another. For example, consider reading different data types from a file descriptor fd (writes would be entirely equivalent):

```
char c, s[N];
int i, v[N];
// 1 char:
read(fd, &c, sizeof(char));
// N char's:
read(fd, s, N*sizeof(char));
// 1 int:
read(fd, &i, sizeof(int));
// N int's:
```

*read(fd, v, N*sizeof(int));*

Examples

1. The program copie-std.c creates a file called "fichier" and copies the data read from the keyboard into it. Input/Output operations are performed using low-level system calls without using the C standard library "stdio.h".

Code II.1 copie-std.c

```
#include <unistd.h>
#include <fcntl.h>
#define size 80
int main()
{
   int fd, nbchar;
   char buf[size];
   // write to a file
   fd = open("file", O_CREAT | O_WRONLY);
   if (fd == -1)
   {
      write(2, "Error opening\n", 14);
      return 1;
   }
   write(1, "Open successfully\n", 19);
   // copy the data entered from
   // the keyboard into the file
   while ((nbchar = read(0, buf, size)) > 0)
      if (write(fd, buf, nbchar) == -1)
         return 1;
   return 0;
}
```

2. The program copie.c copies one file into another.

The execution of copie.c shows that we can copy any type of files, even executable files. However, this version of the program has overwritten the mode (rwxr-xr-x has been transformed into rw-r--r--):

*leibnitz> ls -l stat**
*-rwxr-xr-x 1 jmtorres prof 15926 Nov 15 14:53 stat**
leibnitz> copie stat stat2
done
*leibnitz> ls -l stat**
*-rwxr-xr-x 1 jmtorres prof 15926 Nov 15 14:53 stat**

-rw-r--r-- 1 jmtorres prof 15926 Nov 15 15:12 stat2
leibnitz>

Code II.2 Copie.c

```
#include <sys/wait.h> // wait
#include <fcntl.h>    // les modes
#include <unistd.h>   // les appels système
#define TAILLE 4096

int main(int argc, char *argv[])
{
   int status, src, dst, in, out;
   char buf[TAILLE];
   if (fork() == 0)
   {
      if (argc != 3)
         exit(1);
      src = open(argv[1], O_RDONLY);
      if (src < 0)
         exit(2);
      dst = open(argv[2], O_RDWR | O_CREAT, 0666);
      if (dst < 0)
         exit(3);
      while (1)
      {
         in = read(src, buf, TAILLE);
         if (in < 0)
            exit(4);
         if (in == 0)
            break;
         out = write(dst, buf, in);
         if (out <= 0)
            exit(5);
      }
      close(src);
      close(dst);
      exit(0);
   }
   else
   {
      if (wait(&status) > 0 && (status >> 8) == 0)
         write(1, "done\n", 5);
      else
         write(2, "error\n", 6);
      return 0;
   }
}
```

Seeking to a Specific Position in the File :

#include <unistd.h>
int lseek(int fd, int offset, int whence);

The lseek() system call allows to modify the current position of the file pointer. It returns the new position of the pointer or -1 in case of an error. The positioning within a descriptor is stored in the File Table and not in the Descriptor Table. The second argument, offset, provides the new value for the position. The third argument, whence, can take the following values:

- SEEK_SET: Position = offset.
- SEEK_CUR: Current position + offset.
- SEEK_END: File size + offset. Observe the diagram in FIG II.3.

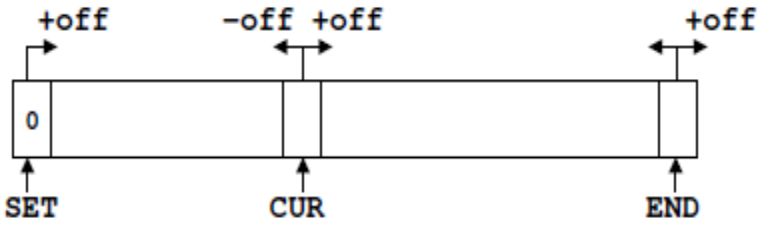


FIG II.3 File Pointer: lseek()

Thus, to determine the current position, one can simply use:

lseek(fd,0,SEEK_CUR).

Let's consider the file "archivo" containing the string: "le printemps". Upon opening this file, the file pointer is positioned at the first character "l". Its value is 0.

```
char buf;
// move the file pointer to position 3 in the file
lseek(fd, 3, SEEK_SET);
// the value of the pointer is now 3.
// read one character into buf (buf receives 'p')
read(fd, &buf, 1);
// move the pointer two characters further
lseek(fd, 2, SEEK_CUR);
// the value of the pointer is now 4 + 2.
// read one character into buf (buf receives 'n')
read(fd, &buf, 1);
// move the pointer to the end of the file
lseek(fd, 0, SEEK_END);
// reading from here will return 0
read(fd, &buf, 1); // will return 0
```

Example : Direct Access to File Elements with seek.c:

Code II.3 seek.c

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   int PLine[200];
   int fd, ncar, i;
   int nline = 0, pfich = 0;
   char buffer[4096];
   // open the file for reading only
   fd = open(argv[1], O_RDONLY);
   if (fd == -1)
      exit(1); // error opening
   PLine[0] = 0; // position of line 0
   while (1)
   { // read the file
      ncar = read(fd, buffer, 4096);
      if (ncar == 0)
         break; // end of file
      if (ncar == -1)
         exit(1); // read error
      for (i = 0; i < ncar; i++)
      {
         pfich++;
         // end of line encountered
         if (buffer[i] == '\n')
            PLine[++nline] = pfich;
      }
   }
   PLine[nline + 1] = pfich;
   for (i = 0; i <= nline; i++)
      printf("PLine[%d] = %d\n", i, PLine[i]);
   // access the first line using seek
   lseek(fd, PLine[0], SEEK_SET);
   ncar = read(fd, buffer, PLine[1] - PLine[0]);
   // display the contents of the buffer to the screen
   write(1, buffer, ncar);

   return 0;
}
```

```
leibnitz> cat >> example
name
first name
code
leibnitz> cat example
name
first name
code
```

```
leibnitz> ./seek example
PLine[0] = 0
PLine[1] = 4
PLine[2] = 15
PLine[3] = 21
name
```

Duplication of file descriptors:

Each process has a file descriptor table. This table contains the file descriptors of the process, where each descriptor points to an open file for the process. When a process is created, the system automatically opens three files for it:

- Standard input is associated with descriptor 0.
- Standard output is associated with descriptor 1.
- Standard error output is associated with descriptor 2.

If the process explicitly opens a file, the descriptor associated with the file will be 3 and so on.

int dup(int oldfd);
int dup2(int oldfd, int newfd);

The dup() system call allows associating multiple descriptors with the same file. dup() associates the smallest available descriptor with the file pointed to by the oldfd descriptor. The dup2() system call associates the descriptor newfd with the file pointed to by oldfd.

Example : File Descriptor Duplication:

Code II.4 dup.c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
   int fd1, fd2, fd3;
   fd1 = open("test", O_CREAT | O_RDWR,0644);
   if (fd1 == -1) {
      perror("open");
      exit(1);
  }
printf("fd1 = %d\n", fd1);
   write(fd1, "hello", 5);
   fd2 = dup(fd1); // fd2 and fd1 are associated
with the file "test"
   printf("fd2 = %d\n", fd2);
   write(fd2, "wor", 3);
   close(0);
 fd3 = dup(fd1); // fd3 and fd1 are associated
                    //with the file "test"
   printf("fd3 = %d\n", fd3);
   write(0, "ld", 2);
   dup2(3, 2);
   write(2, "!\n", 2);
   return 0;
}
```

Exécution of dup.c :

leibnitz> gcc -o dup dup.c
leibnitz> dup
fd1 = 3
fd2 = 4
fd3 = 0

leibnitz> ls -l test
-rw-r--r-- 1 jmtorres prof 13 Nov 15 16:03 test
leibnitz> more test
hello world!
leibnitz>

You should remember that if two file descriptors are associated with the same entry in the table, moving the pointer with lseek() (or with read/write) will affect the other. Observe the small program lseek.c:

Code II.5 lseek.c

```
# include <unistd.h>
int main ( void )
{
int fd ;
if ( ( fd = open ( "file " , O_RDWR) ) < 0 )
exit (1) ;
if ( lseek ( fd , 10 , SEEK_SET ) < 0 )
   exit (1) ;
if ( write ( fd , "123" , 3 ) < 0 )
    exit (1) ;
return 0 ;
}
```

Unsurprisingly, the execution on a file named "file":

pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> lseek
pascal> cat file
ABCDEFGHIJ123NOPQRSTUVWXYZ
pascal>

But in this other program dup-lseek.c, two different descriptors with the same entry in the file table will modify the file traversal pointer:

Code II.6 dup-lseek.c

```
#include <unistd.h>
int main(void) {
    int fd;
    if ((fd = open("file", O_RDWR)) < 0)
        exit(1);
    dup2(fd, 1);
    if (lseek(fd, 10, SEEK_SET) < 0)
        exit(1);
    if (lseek(1, 5, SEEK_CUR) < 0)
        exit(1);
    if (write(fd, "123", 3) < 0)
        exit(1);
    return 0;
}
```

Execution of dup-lseek.c on a file:

pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup-lseek
pascal> cat file
ABCDE123IJKLMNOPQRSTUVWXYZ
pascal>

The first lseek sets the pointer to the absolute position 10, but the second lseek (despite using a different fd, it uses the same entry in the table) moves it back 5 bytes. Finally, look at the following example, dup3.c, which opens the same file "file" twice:

Code II.7 dup3.c

```
#include <unistd.h>
void main(void)
{
   int fd1, fd2;
   if ((fd1 = open("file", O_RDWR)) < 0)
      exit(1);
   if ((fd2 = open("file", O_WRONLY)) < 0)
      exit(1);
      if (lseek(fd1, 10, SEEK_SET) < 0)
      exit(1);
   if (write(fd1, "456", 3) < 0)
      exit(1);
   if (lseek(fd2, 5, SEEK_CUR) < 0)
      exit(1);
   if (write(fd2, "123", 3) < 0)
      exit(1);
   return 0;
}
```

Output

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup3
pascal> cat file
ABCDE123IJ456NOPQRSTUVWXYZ
pascal>
```

Why does this behavior occur? Remember that two openings of the same file provide two filedescriptors, with independent entries in the file table.

Fetching File Information:

int stat(char filename, struct stat * buf);*

The stat() system call retrieves information about the file specified by filename. Upon success, the stat() system call returns 0, and the file information is stored in buf. In case of an error, it returns -1. stat() stores the retrieved information in a structure of type stat:

```
struct stat {
   mode_t st_mode;    /* Protection code */
   ino_t st_ino;      /* File's inode number */
   dev_t st_dev;      /* Device */
   nlink_t st_nlink;  /* Number of hard links */
   uid_t st_uid;      /* Owner's user ID */
   gid_t st_gid;      /* Owner's group ID */
   off_t st_size;     /* File size (in bytes) */
   time_t st_atime;   /* Last access time */
   time_t st_mtime;   /* Last modification time */
   time_t st_ctime;   /* Last status change time */
};
```

buf.st_ino indicates, for example, the inode number of the file stored in buf. There are predefined macros that take buf.st_mode as an argument and return 1 or 0:

- S_ISDIR(buf.st_mode): true if the file is a directory.
- S_ISCHR(buf.st_mode): true if it is a character special file.
- S_ISBLK(buf.st_mode): true if it is a block special file.
- S_ISREG(buf.st_mode): true if it is a regular file.
- S_ISFIFO(buf.st_mode): true if it is a pipe or FIFO.

The following code allows you to retrieve information from a file.

Code II.8 stat.c

```
#include <sys/stat.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
   struct stat buf;
   mode_t mode;
   int result;
   result = stat(argv[1], &buf);
   if (result == -1)
    printf("Information about %s not available\n", argv[1]);
   else {
      mode = buf.st_mode;
      if (S_ISDIR(mode))
         printf("%s is a directory ", argv[1]);
      else if (S_ISREG(mode))
         printf("%s is a regular file ", argv[1]);
      else if (S_ISFIFO(mode))
         printf("%s is a pipe ", argv[1]);
      else
         printf("%s is a special file ");
      printf("\n File size: %ld\n", buf.st_size);
   }
   return 0;
}
```

Exécution of stat.c :

leibnitz> stat
Information about (null) not available
leibnitz> stat stat.c
stat.c is a regular file
File size: 775
leibnitz>

The following program, estado.c, displays all the attributes of a list of files. Note the use of the functions getpwuid() and getgrgid() to obtain the user's UID and group.

Code II.9 estado.c

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <pwd.h>
# include <grp.h>
char permisos[] = {'x', 'w', 'r'};
void estado (char archivo[]);
int main (int argc, char *argv[])
{
   int i;
   if (argc != 2)
   {
      printf("Usage: %s file_name\n", argv[0]);
      exit(1);
   }
   for (i = 1; i < argc; i++)
      estado(argv[i]);
   return 0;
}
void estado(char archivo[])
{
   struct stat buf;
   struct passwd *pw;
   struct group *gr;
   int i;
   stat(archivo, &buf);
   printf("Archivo: %s\n", archivo);
   printf("Dispositivo: %d, %d\n",
     (buf.st_dev & 0xff00) >> 8, buf.st_dev & 0x00ff);
   printf("Inode: %d\n", buf.st_ino);
   printf("Type: ");
   switch (buf.st_mode & S_IFMT)
   {
   case S_IFREG:
      printf("regular\n");
      break;
   case S_IFDIR:
      printf("directory\n");
      break;
   case S_IFCHR:
      printf("special character\n");
      break;
   case S_IFBLK:
      printf("special block\n");
      break;
   case S_IFIFO:
      printf("FIFO\n");
      break;
   }
   printf("Permissions: 0%o, ", buf.st_mode & 0777);
   for (i = 0; i < 9; i++)
   {
      if (buf.st_mode & (0400 >> i))
         printf("%c", permisos[(8 - i) % 3]);
      else
         printf(" ");
   }
   printf("\n");
   printf("Links: %d\n", buf.st_nlink);
   printf("UID: %d\n", buf.st_uid);
   printf("Name: ");
   pw = getpwuid(buf.st_uid);
   printf("%s\n", pw->pw_name);
   printf("GID: %d\n", buf.st_gid);
   printf("Name: ");
   gr = getgrgid(buf.st_gid);
   printf("%s\n", gr->gr_name);
   switch (buf.st_mode & S_IFMT)
   {
   case S_IFCHR:
   case S_IFBLK:
      printf("Device: %d, %d\n",
           (buf.st_rdev & 0xff00) >> 8, buf.st_rdev &
0x00ff);
   }
   printf("Length: %d bytes.\n", buf.st_size);
   printf("Last access: %s ",
        asctime(localtime(&buf.st_atime)));
   printf("Last modified: %s ",
        asctime(localtime(&buf.st_mtime)));
   printf("Last status change: %s ",
        asctime(localtime(&buf.st_ctime)));
}
```

Execution of estado.c :

```
leibnitz> gcc -o estado estado.c
leibnitz> estado estado
 File                 : estado
Device               : 0, 0
 i-node              : 1854250
Type                 : regular
Permissions          : 0755, rwxr-xr-x
Links                : 1
UID                  : 11047
Name                 : jmtorres
GID                  : 100
Name                 : prof
Length               : 17555 bytes.
Last access          :Tue Dec 3 15:03:24 2024
Last modification    :Tue Dec 3 15:03:24 2024
Last status change: Tue Dec 3 15:03:24 2024
leibnitz>
```

Delete a file:

```
#include <unistd.h>
int unlink(char* nom_fichier);
```

The **unlink()** system call is used to delete a file. It removes the link between the **filename** and the corresponding inode. It returns 0 if the file was successfully deleted or -1 if an error occurred. The following example demonstrates that even if a file is deleted, its content is still accessible as long as it is not closed.

Example 11. Deleting (and reusing) a file.

Code II.10 unlink-cs.c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#define N 16

int main(void) {
   char chaine[N];
   int fp;
   write(1, "Création fichier\n", 17);
   fp = open("test", O_RDWR | O_CREAT, 0666);
   if (fp < 0) {
      exit(1);
   }
   write(fp, "0123456789abcdef", N);
   system("ls -l test");
   write(1, "Effacement fichier\n", 19);
   if (unlink("test") == -1) {
      perror("unlink");
      exit(1);
   }
   system("ls -l test");
   write(1, "Relecture du contenu du fichier\n", 32);
   lseek(fp, 0, SEEK_SET);
   read(fp, chaine, N);
   write(1, chaine, N);
   write(1, "Fermeture fichier\n", 18);
   close(fp);
   return 0;
}
```

Exécution of unlink-cs.c :

leibnitz> gcc -o unlink unlink-cs.c
leibnitz> unlink
Creating file
-rw-r--r-- 1 jmtorres prof 26 Nov 1 17:17 test
Deleting file
ls: test: No such file or directory
Reading file content again
Read: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Closing file

Fast Data Transfer

The conventional way of copying a file to another involves allocating a buffer of fixed size, copying some data into the buffer, and then writing the buffer to another descriptor, repeating this process until all the data is copied. This is neither efficient in time nor space as it involves additional memory usage (the buffer) and extra data copying. The sendfile() system call provides an efficient mechanism for copying files where the buffer is eliminated.

#include <sys/sendfile.h>
sendfile(write_fd, read_fd, &offset, stat_buf.st_size);

The descriptors can be files, sockets, or other devices. The parameters are **write_fd**, the file descriptor to write to, **read_fd**, the read descriptor, **offset**, a pointer to a dynamic offset, and **st_size**, the number of bytes to copy. You can use **fstat()** to determine this size. The **offset** contains the offset (0 indicates the start of the file). The return value is the number of bytes copied. The program fcopie.c demonstrates this efficient technique.

Code II.11 fcopie.c

```
# include <fcntl.h>
# include <sys/sendfile.h>
# include <sys/stat.h>
# include <sys/types.h>
# include <unistd.h>
int main ( int argc , char
 argv [ ] ){
int fd_r ;
 int fd_w;
struct stat buf ;
off_t offset = 0 ;
// source file descriptor
fd_r = open ( argv [ 1 ] , O_RDONLY) ;
// get its size
fstat ( fd_r , & buf ) ;
// output file descriptor with the same permissions as the source fd
fd_w = open ( argv [ 2 ] , O_WRONLY|O_CREAT, buf . st_mode ) ;
```

```
// Copying
sendfile ( fd_w , fd_r , & offset , buf . st_size ) ;
close ( fd_r ) ;
close ( fd_w ) ;
return 0 ;
}
```

II.4 Directories

Modern operating systems adopt a tree-like structure to represent the file system. The nodes of the tree are directories, and the leaves are files. A directory consists of files and subdirectories. To access a file, you only need to specify the directories along the path from the root of the tree to the file (path). In the Unix/Linux system, each directory also contains its own reference "." and that of the parent directory "..". Table I.2 illustrates a Unix directory. A directory is also considered as a special file composed of files.

Table II.2 Unix directories

i-node	File name
20	.
3	..
100	Chap1
2378	Chap2
125	scheduler
⋮	

II.5 POSIX Services on Directories

Directories in Unix/Linux have an entry for each file. Each entry contains the file name and the inode number. The Unix/Linux system offers POSIX system calls to manipulate directories:

- mkdir(): create a directory.
- rmdir(): remove an empty directory.
- opendir(): open a directory.
- closedir(): close a directory.
- readdir(): read entries from a directory.
- rewinddir(): reset the directory pointer.
- link(): create an entry in a directory.
- unlink(): remove an entry from a directory.
- chdir(): change the working directory.

- rename(): rename a directory.
- getcwd(): get the current directory name.

Creating a Directory

#include <sys/types.h>
#include <dirent.h>
*int mkdir(const char *name, mode_t mode);*

The mkdir() system call creates a directory with the UID of the owner = effective UID and the GID of the owner = effective GID. It takes the name of the directory as the name argument and the protection bits as the mode argument. It returns 0 or -1 in case of an error.

Removing a directory

#include <sys/types.h>
*int rmdir(const char *name);*

rmdir() deletes the directory if it is empty. If the directory is not empty, it is not deleted. Arguments: name is the directory name. It returns 0 or -1 in case of an error.

Opening a directory

#include <sys/types.h>
#include <dirent.h>
*DIR *opendir(char *nom);*

opendir() opens a directory as a sequence of entries. It sets the pointer to the first element. opendir() receives name as an argument, which is the directory name. It returns a pointer of type DIR to be used in readdir() or closedir(), or NULL if there was an error.

Closing a directory

#include <sys/types.h>
#include <dirent.h>
*int closedir(DIR *dirp);*

The closedir() system call closes the link between dirp and the directory entry sequence. Arguments: dirp is the pointer returned by opendir(). It returns 0 or -1 in case of an error.

Reading directory entries

#include <sys/types.h>
#include <dirent.h>
*struct dirent *readdir(DIR *dirp);*

The readdir() system call receives dirp as an argument, which is the pointer returned by opendir(). It returns a pointer to an object of type struct dirent representing a directory entry, or NULL if there was an error. It returns the next entry of the directory associated with dirp

and advances the pointer to the next entry. The structure is implementation-dependent. For simplification, it can be assumed that a member char *d_name is obtained.

Change a directory

*int chdir(char *name);*

chdir() modifies the current directory from which relative names are formed. Arguments: name is the name of a directory. It returns 0 or -1 in case of an error.

Rename a directory

#include <unistd.h>
*int rename(const char *old, const char *new);*

rename() changes the name of the directory old to new. Arguments: old is the name of an existing directory, and new is the new name of the directory. It returns 0 or -1 in case of an error. The code liste-dir.c shows an example of traversing a Directory.

Code II.12 liste-dir.c

```
# include <sys/types.h>
# include <dirent.h>
# include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAX_BUF 256

int main (int argc , char *argv[]) {
   DIR *dirp;
   struct dirent *dp;
   char buf[MAX_BUF];
   // Show the current directory
   getcwd(buf, MAX_BUF);
   printf("Current directory: %s\n", buf);
   // Open the passed directory as an argument
   dirp = opendir(argv[1]);
   if (dirp == NULL) {
      printf("Cannot open %s\n", argv[1]);
   }
   else {
      // Read entry by entry
      while ((dp = readdir(dirp)) != NULL)
         printf("%s\n", dp->d_name);
      closedir(dirp);
   }
   exit(0);
}
```

Execution of liste-dir.c :

```
leibnitz> pwd
/home/ada/users/jmtorres/inf3600/logiciel
leibnitz> ls
bin/ chap2/ chap4/ chap7/ examen/ scheduler/
tmp/
chap1/ chap3/ chap5/ chap9/ parallel/ sockets/
leibnitz> chap9/liste-dir ../logiciel
Current directory:
/home/ada/users/jmtorres/inf3600/logiciel
.
..
chap1
chap2
chap3
scheduler
bin
tmp
parallel
chap4
sockets
chap5
chap7
examen
chap9
leibnitz>
```

II.6 Directories on Unix System V

In Unix System V, the getdents() system call allows reading an entry from the current directory and inserts the information about this entry into buf.

#include <sys/dirent.h>
*int getdents(int fd, struct dirent *buf, int size);*

The dirent structure consists of the following fields:

- buf.d_ino: it is the inode number.
- buf.d_off: it is the relative position of the next directory entry.
- buf.d_reclen: it is the length of the directory entry structure.
- d_name: it is the name of the file.

The Code getd.c shows an example of Traversing a Directory:

Code II.12 gestd.c

```
# include <stdio.h> // printf
# include <string.h> // strcmp
# include <fcntl.h> // O_RDONLY
# include <dirent.h> // getdents
# include <sys/types.h> // mode_t
# include <sys/stat.h> // macros S_ISDIR
# include <unistd.h> // open ...
int main ( int argc , char argv[] )
 {
   int fd, nbcar, result;
   struct stat buf;
   mode_t mode;
   struct dirent entree;
   result = stat(argv[1], &buf);
   if (result == 1)
   {
      printf("Infos on %s not available\n", argv[1]);
      exit(1);
   }
   mode = buf.st_mode;
   if (S_ISDIR(mode))
      printf("%s is a directory\n ", argv[1]);
   else
   {
      printf("%s is not a directory\n ", argv[1]);
      exit(1);
   }
   fd = open(argv[1], O_RDONLY);
   if (fd == 1)
   {
      printf("Error opening dir. %s\n", argv[1]);
      exit(1);
   }
   lseek(fd, 0, SEEK_SET);
   while (1)
   {
      nbcar = getdents(fd, &entree, 40);
      if (nbcar == 1)
      {
         printf("Error reading dir. %s\n", argv[1]);
         exit(1);
      }
      if (nbcar == 0)
      {
         printf("End of reading dir. %s\n ", argv[1]);
         break;
      }
      if (strcmp(entree.d_name, ".") != 0 &&
strcmp(entree.d_name, "..") != 0)
         printf("%s\n ", entree.d_name);
      lseek(fd, entree.d_off, SEEK_SET);
   } // end of while
   close(fd);
   printf("\nChanging working directory\n");
   chdir(argv[1]);
   system("pwd");
   printf("Contents of new working directory: \n");
   system("ls");
   exit(0);
}
```

jupiter% gcc getd.c -o getd
jupiter% getd newdir
newdir is a directory
example
test
End of reading dir. newdir
Changing working directory
/jupiter/home/bouchene/gestfichier/newdir
Contents of new working directory:
example test

Chapter III

FILE STORAGE AND SHARING

III.1 Introduction

Data files on disks are distributed in blocks of fixed size. Reading or writing an element of a file will therefore involve transferring the entire block that contains that element. For fast access, it is therefore advisable to use large block sizes. However, files, including files of 1 byte, have a minimum size of 1 block. If a disk contains many small files and the blocks are large, the wasted space will be considerable. Therefore, adequate file storage techniques need to be implemented. There are three ways to allocate files: contiguous allocation, linked list of blocks, and inode-based indexing.

III.2 File Storage

Studies on many systems have shown that the average file size is 1 KB. Typically, commonly used block sizes are 512, 1024, or 2048 bytes. Each disk retains, in one or more specific blocks, a number of pieces of information such as the number of blocks, their size, their states, among others. Each file corresponds to a list of blocks containing its data. Allocation is generally non-contiguous, so the blocks are distributed quasi-randomly on the disk. Files retain all their blocks using two methods: linked list and inode-based indexing.

III.2.1 Contiguous Allocation

The file allocation table contains only one entry per file, with the starting block and the file size. Contiguous allocation is simple and efficient but has the serious drawback of quickly filling space with unusable zones (due to external fragmentation), and in this case, compaction is not feasible within reasonable timeframes. Additionally, the file size must be declared at the time of creation.

III.2.2 Linked List

The blocks of the same file are linked under a linked list representation shown in Figure 12.1. Each block will contain data as well as the address of the next block. The file must memorize the number of the first block. For example, if a block contains 1024 bytes and

if the number of a block is coded on 2 bytes, 1022 bytes will be reserved for data and 2 bytes for linking to the next block.

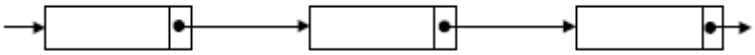


FIG III.1 linked list

This method makes random access to elements of a file particularly inefficient when used as is. Indeed, to access an element in block n of a file, the system will have to traverse the preceding n-1 blocks. The MS-DOS system utilizes an enhanced version of linked lists. It keeps the first block of each file in its directory. It then optimizes access to subsequent blocks by storing their references in a File Allocation Table (FAT). Each disk has a FAT table, and it has as many entries as there are blocks on the disk. Each entry in the FAT contains the number of the next block. For example, in the following table:

Table III.1 Structure of FAT

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	x	EOF	13	2	9	8	L	4	12	3	E	EOF	EOF	L	...

"x" indicates the size of the disk, "L" denotes a free block, and "E" represents a damaged block. The file starting at block 6 will consist of the blocks: 6-->8-->4-->2. The traversal of the FAT is significantly faster than the block chain. However, it must constantly be entirely in memory to avoid disk accesses for locating a block.

III.2.3 Inode Table

In the Unix system, each file is associated with a unique identification number. Each identification number is associated with a set of information called an inode. Among the fields of the inode, the index table indicates the physical location of the file. It consists of 13 entries. The first 10 entries contain the numbers of the first 10 blocks composing the file. For files larger than 10 blocks, indirect blocks are used. Block number 11 contains the number of a block composed itself of data block addresses. If the blocks have a size of 1024 bytes and if they are numbered on 4 bytes, block number 11 can designate up to 256 blocks. In total, the file using single indirect addressing can have a maximum size of 266 blocks. Similarly, block number 12 contains a double indirect pointer, and block number 13 contains a triple indirect pointer. Figure 12.2 shows the usage of single, double, and triple indirections of an inode.

A file can have a maximum size of 16 GB when using triple indirection pointers. The inode table has up to 4 levels of indexing, but only the first level is present in memory when the corresponding file is open.

In Linux, the inode table differs slightly from that used in Unix System V. In Linux, the table consists of 12 entries for blocks and 3 entries for indirections.

III.3 The /proc File System

Linux includes the pseudo-file system /proc. This is actually a directory that contains entries numbered by the current processes. The entries contain information in a human-readable format.

III.3.1 System Statistics

The /proc/uptime entry contains information in seconds about the system's uptime and idle time:

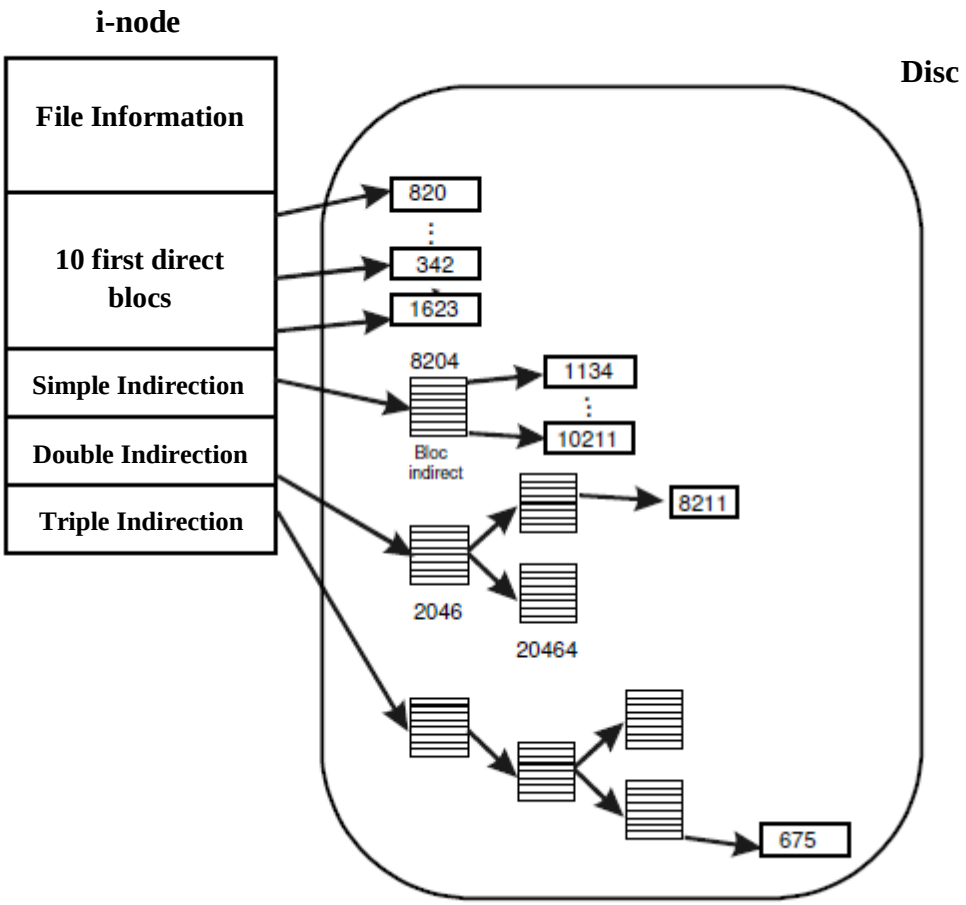


FIG III.2 Inode Indirections.

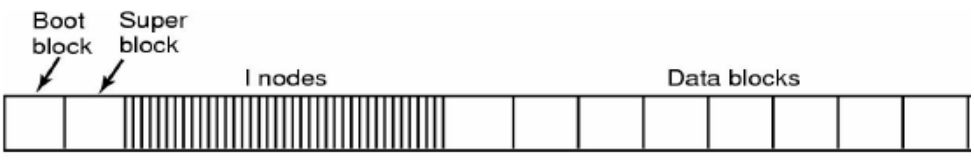


FIG III.3 Inode and Superblock

leibnitz> cat /proc/uptime
547217.94 288458.71

III.3.2 Environment Variable

The entry of /proc/environ contains information about the environment. The following program shows the environment variable obtained from the /proc entry.

Code III.1 env.c

```
# include <fcntl.h>
# include <stdio.h>
# include <stdlib.h>
# include <sys/stat.h>
# include <sys/types.h>
# include <unistd.h>
int main ( int argc , char argv [ ] )
{
int pid = atoi ( argv [ 1 ] ) ;
int fd ;
char filename [ 24 ] ;
char environment [ 8192 ] ;
size_t length ;
char next_var ;
// Generate the environment name for the process
snprintf ( filename , sizeof ( filename ) , "/proc/%d/environ " , ( int ) pid ) ;
// Read the file
fd = open ( filename , O_RDONLY) ;
length = read ( fd , environment , sizeof ( environment ) ) ;
close ( fd ) ;
environment [ length ] = '\0' ;
// Cycle through the variables
next_var = environment ;
while ( next_var < environment + length )
{
printf ( "%s\n" , next_var ) ;
next_var += strlen ( next_var ) + 1 ;
}
return 0 ;
}
```

For example, in a window, the "cat" command is launched, which remains blocked waiting for an argument from the user.

leibnitz> cat

...

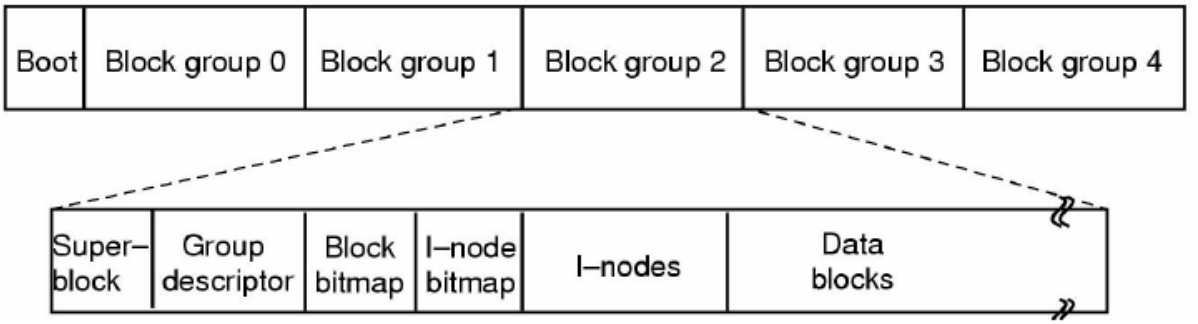


FIG III.4 EXT2 Partition (above) and an EXT2 Block Group (below).

In another terminal, we check the processes and run the program env.c with the process number of cat:

leibnitz> ps -u jmtorres
PID TTY TIME CMD
1956 pts/4 00:00:00 tcsh
5764 pts/4 00:00:00 cat
5768 pts/1 00:00:00 tcsh
5819 pts/1 00:00:00 ps
leibnitz> gcc -o env env.c
leibnitz> env 5764
REMOTEHOST=torres.dgi.polymtl.ca
TERM=ansi
HZ=100
HOME=/home/ada/users/jmtorres
SHELL=/bin/tcsh
PATH=/home/ada/users/jmtorres/bin:/home/ada/users/jmtorres/
home/ada/users/jmtorres/office52:/home/ada/users/jmtorres/
bin/X11:/usr/local/bin:/usr/local/sbin:/usr/local/qt/bin:
/usr/openwin/bin:/usr/X11/bin:/usr/share/texmf/bin:/usr/bin:/
usr/local/java/j2sdk1.4.0_01/bin:/bin:/usr/bin:/sbin:/usr/games:/
usr/local/Simscript/bin:/usr/local/:/usr/local:/usr/local/Eclips
e/bin/i386_linux
...
HOSTNAME=leibnitz
...
EDITOR=emacs
...
CONSOLE=OFF
leibnitz>

III.3.3 File Descriptor Process

The fd entry is a subdirectory containing entries for the files descriptors opened by a process. Each entry is a symbolic link to the file or device opened in that file descriptor. You can write to or read from these symbolic links; write to or read from the corresponding file or device opened by the process. Entries in the fd subdirectory are named by the file descriptor numbers. We present an elegant strategy that can be tried with the fd entries in /proc. Open a shell window and find the process ID of the shell process by running ps:

```
bash-2.05a$ ps
PID TTY TIME CMD
1956 pts/4 00:00:00 tcsh
2413 pts/4 00:00:00 bash
2414 pts/4 00:00:00 ps
bash-2.05a$
```

In this case, the shell (bash) is running under process 2413. Now, open a second window and view the contents of the fd subdirectory for this process:

```
bash-2.05a$ ls -l /proc/2413/fd
total 0
lrwx------ 1 jmtorres prof 64 Nov 11 15:53 0 -> /dev/pts/4
lrwx------ 1 jmtorres prof 64 Nov 11 15:53 1 -> /dev/pts/4
lrwx------ 1 jmtorres prof 64 Nov 11 15:53 2 -> /dev/pts/4
lrwx------ 1 jmtorres prof 64 Nov 11 15:53 255 -> /dev/pts/4
bash-2.05a$
```

The file descriptors 0, 1, and 2 are initialized to standard input, standard output, and standard error respectively. Therefore, writing to /proc/2413/fd/1 writes to the device attached to stdout by the shell process—in this case, the pseudo TTY of the first window—and in the second window, a message is written to this file:

```
bash-2.05a$ echo "Hola..." >> /proc/2413/fd/1
bash-2.05a$
```

The text "Hola..." then appears on the first window.

III.4 File Sharing

When multiple users work on the same project, they often need to share files. To facilitate sharing of an existing file, Unix/Linux systems offer the possibility of associating multiple references to a file by creating hard links or symbolic links. For example, in Figure II.5, file X is shared between the directories /B/B/X and /C/C/C/X.

III.4.1 Hard Links

To share a file located on a disk, you can associate multiple paths or references with its inode number. This type of link is physical. Physical links are possible only if the paths refer exclusively to directories managed by the same file system. In other words, you cannot create a link from a directory on one machine to a file located on another machine. The system call used to create a physical link to an existing file is link():

int link(char oldpath, char* newpath);*

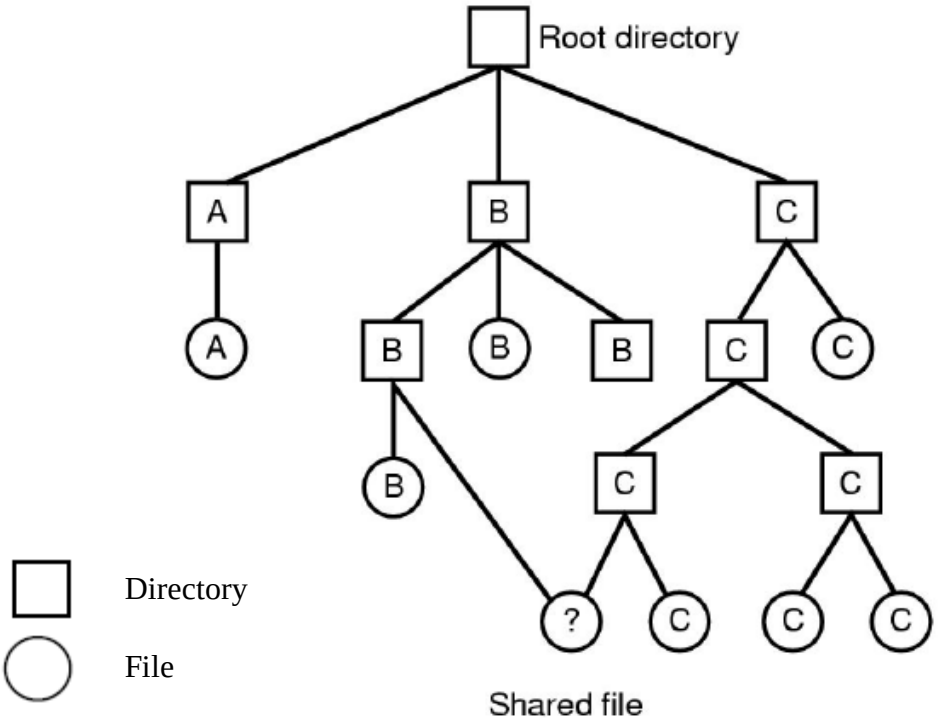


Fig III.5 shared files

The equivalent shell command to link() is ln:

ln oldpath newpath

The *link()* system call associates another path *newpath* with a file that has the path *oldpath*. Both paths refer to the same file (*newpath* and *oldpath* reference the same file). Once the links are established, the file can be referenced by either of the two names (paths). For example, the *link("origref", "../autreref")* system call operates as follows:

-It searches in the current directory for the entry with the file name "origref" and retrieves the inode number. It increments the reference count of the inode.

- It adds an entry in the parent directory of the current directory with the file name "autreref" and the inode number of the file "origref".

There is a system call that allows removing a physical link to a file. It's called unlink:

*int unlink(char *path);*

unlink() removes the reference path to a file. If the file no longer has any references, it is deleted from the disk. For example, unlink("origref") works as follows:

-It locates, in the current directory, the entry with the file name "origref". It retrieves the inode number and then removes this entry from the directory.
-It decrements the number of references to the inode. If the number of references becomes equal to 0, the file is deleted from the disk (data blocks and inode).

The shell command that allows removing a link is rm. Look at the following example:

pascal> cat original
example of a file with multiple references.
pascal> ln original otherref
pascal> ls -l original otherref
-rw-r--r-- 2 jmtorres prof 48 Nov 18 15:06 otherref
-rw-r--r-- 2 jmtorres prof 48 Nov 18 15:06 original
pascal> cat otherref
example of a file with multiple references.
pascal> rm original
pascal> ls -l original otherref
/bin/ls: original: No such file or directory
-rw-r--r-- 1 jmtorres prof 48 Nov 18 15:06 otherref
pascal> cat otherref
example of a file with multiple references.
pascal>

III.4.2 Symbolic Links

Symbolic links allow creating links to files that are not necessarily managed by the same file system. They have the advantage of being able to create links to files located on any remote computer. In addition to the local path, the network address of the machine needs to be included in the path. A symbolic link is an indirect pointer to an existing file. Deleting a symbolic link to a file does not affect the file itself. The Unix shell command that creates symbolic links is ln with the -s option for symbolic. For example, the command:

ln -s /usr/include/stdio.h stdio.h

creates a new entry in the current directory for a new file named stdio.h. The content of this new file is the path:

/usr/include/stdio.h.

The type of the created file is a link (l).

For example:

pascal> ln -s /usr/include/stdio.h stdio.h

pascal> ls -l stdio.h

lrwxrwxrwx 1 jmtorres prof 20 Nov 18 15:02 stdio.h -> /usr/include/stdio.h

pascal> cat stdio.h

...

This command displays the content of the file */usr/include/stdio.h*

Issues: Links allow files to have multiple access paths. Programs that traverse all directories to process files may encounter files shared multiple times. The same file may undergo the same treatment multiple times (for example, they may be copied multiple times). Once the links are established, multiple users can access the same file. Accesses can be concurrent. The Unix system provides system calls to control access to data in the same file.

Reading Symbolic Links

The readlink() system call retrieves the target of a symbolic link. It takes as arguments the path to the symbolic link, a buffer to receive the name of the target, and the length of this buffer. readlink does not append a null character at the end of the path name. However, it returns the number of characters in the path. If the first argument to readlink points to a file that is not a symbolic link, readlink sets errno to EINVAL and returns -1.

The program *sym.c* prints the path of a specified symbolic link provided on the command line:

Code III.2 sym.c

```
# include <errno.h>
# include <stdio.h>
# include <unistd.h>

int main (int argc, char *argv[]) {
   char target_path[256];
   char *link_path = argv[1];
   // Attempt to read the path of the symbolic link
   int len = readlink(link_path, target_path, sizeof(target_path));
   if (len == 1) { // The call failed
      if (errno == EINVAL)
         printf("%s is not a symbolic link\n", link_path);
      else
         perror("readlink");
      return 1;
   }
```

```
else {
        // Null-terminate the target path
        target_path[len] = '\0';
        // Print the target path
        printf("%s\n", target_path);
        return 0;
    }
}
```

Observe the use of sym.c :

pascal> ln -s /usr/bin/wc my_link
pascal> sym my_link
/usr/bin/wc
pascal>

Here is another example illustrating the differences between symbolic and hard links. We will use the "-li" option (listing of i-nodes) of ls. Let's first demonstrate symbolic links:

pascal> cat > liga.txt
example of a file with multiple references. ^D
*pascal> ls -l liga**
-rw-r--r-- 1 jmtorres prof 48 Nov 18 15:06 liga.txt
pascal> ln liga.txt liga.ln
*pascal> ls -li liga**
3052239 -rw-r--r-- 2 jmtorres prof 48 Nov 18 15:06 liga.ln
3052239 -rw-r--r-- 2 jmtorres prof 48 Nov 18 15:06 liga.txt
pascal> rm liga.txt
pascal> cat liga.ln
example of a file with multiple references.
*pascal> ls -li liga**
3052239 -rw-r--r-- 1 jmtorres prof 48 Nov 18 15:06 liga.ln
pascal>

Please observe now what happens with hard links:

pascal> cp liga.ln ligado.txt
pascal> ln -s ligado.txt liga2.ln
*pascal> ls -li liga**
3052242 lrwxrwxrwx 1 jmtorres prof 10 Nov 18 15:51 liga2.ln ->
ligado.txt
3052240 -rw-r--r-- 1 jmtorres prof 48 Nov 18 15:51 ligado.txt
3052239 -rw-r--r-- 1 jmtorres prof 48 Nov 18 15:06 liga.ln
pascal> rm ligado.txt
pascal> cat liga2.ln
cat: liga2.ln: No such file or directory
*pascal> ls -li liga**

3052242 lrwxrwxrwx 1 jmtorres prof 10 Nov 18 15:51 liga2.ln ->
ligado.txt
3052239 -rw-r--r-- 1 jmtorres prof 48 Nov 18 15:06 liga.ln
pascal>

Code III.3 copie-mem.c

```
# include <sys/types.h>
# include <sys/stat.h>
# include <sys/mman.h>
# include <fcntl.h>
# include <stdio.h>
# include <unistd.h>
void main ( int argc , char
* argv )
{
 int i , fdo , fdd ;
char * org , * dst , p , q ;
struct stat bstat ;
if ( argc !=3 )
{
fprintf( stderr , "%s orig dest\n" , argv [ 0 ] ) ;
exit ( 1 ) ;
}
// Open source for reading
 fdo=open ( argv [ 1 ] , O_RDONLY) ;
// Create destination
fdd=open ( argv [ 2 ]
          ,O_CREAT|O_TRUNC|O_RDWR, 0640 ) ;
// Length of source file
if ( fstat ( fdo , & bstat )<0)
{
perror ( " fstat " ) ;
close ( fdo ) ;
close ( fdd ) ;
unlink ( argv [ 2 ] ) ;
exit ( 1 ) ;
}
// Length of destination = length of source
if ( ftruncate ( fdd , bstat . st_size )<0)
{
perror ( " ftruncate " ) ;
close ( fdo ) ;
close ( fdd ) ;
unlink ( argv [ 2 ] ) ;
exit ( 1 ) ;
 }
// Project the source file
if ( ( org=mmap( ( caddr_t ) 0 , bstat . st_size ,
PROT_READ,
MAP_SHARED, fdo , 0 ) ) == MAP_FAILED)
{
perror ( "mmap source " ) ;
close ( fdo ) ;
close ( fdd ) ;
unlink ( argv [ 2 ] ) ;
exit ( 1 ) ;
 }
// Project the destination file
if ( ( dst=mmap( ( caddr_t ) 0 , bstat . st_size ,
PROT_WRITE ,
MAP_SHARED, fdd , 0 ) ) == MAP_FAILED)
{
perror ( "mmap destination " ) ;
close ( fdo ) ;
close ( fdd ) ;
unlink ( argv [ 2 ] ) ;
exit ( 1 ) ;
 }
close ( fdo ) ;
close ( fdd ) ;
// Copy
p=org ;
q=dst ;
for ( i = 0 ; i<bstat . st_size ; i ++)
*q++= *p++;
 // Eliminate projections
munmap( org , bstat . st_size ) ;
munmap( dst , bstat . st_size ) ;
}
```

Do symbolic links work with directories? If you doubt it, type the following commands:

pascal> ln -s /tmp temporal
pascal> cd temporal
pascal> ls -l
...

and now:

pascal> cd /tmp
pascal> ls -l
...

copie-mem.c performs file copying using Posix system calls and memory-mapped files.

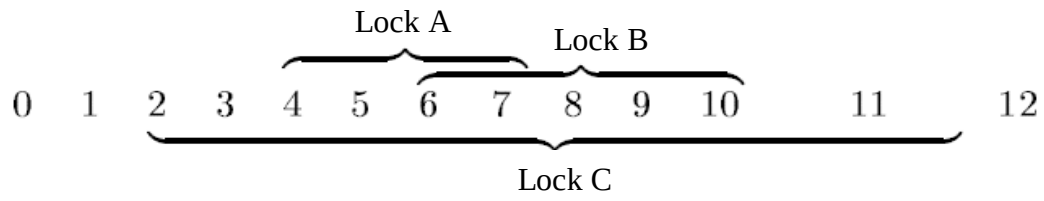
III.5 File Locking

Processes that wish to perform exclusive accesses to a file can lock either a single byte or an entire file in a single atomic operation. Two types of file locking are available: shared and exclusive. If a portion of a file is locked with a shared lock, a second attempt to overlap a shared lock is allowed, but any attempt to place an exclusive lock will be rejected until the lock is released. If a portion of a file contains an exclusive lock, any attempt to lock any portion will be rejected until the lock is released.

III.5.1 Locking a Portion of a File

When a process requests a lock, it must specify whether it wants to be blocked if the lock cannot be obtained. If it chooses to be blocked, when the existing lock is released, the process will be unblocked and its lock can be placed. If it does not want to be blocked, the system call terminates immediately, with a code indicating whether the locking could be performed or not. For example, consider four concurrent processes A, B, C, and D sharing the same file. The processes make the following locking requests in order:

- Process A requests a shared lock on bytes 4 to 7 of the file.
- Process B requests a shared lock on bytes 6 to 9.
- Process C requests a shared lock on bytes 2 to 11.
- Finally, process D requests an exclusive lock on byte 9 with a blocking request in case of failure.



Processes A, B, and C manage to place their shared locks. However, process D is blocked until byte 9 becomes free, i.e., until B and C release their locks.

III.5.2 Posix File Locking Services

Posix has defined the system calls lockf() and fcntl() as standard for locking parts of a file.

lockf()

int lockf(int fd, int function, int size);

The **lockf()** service returns 0 on success and -1 on failure. Here, **fd** is a file descriptor of the file, and the open mode must be **O_WRONLY** or **O_RDWR**. The **function** parameter indicates the action to perform. Possible values for **function** are defined in the **<unistd.h>** header as follows:

- **F_ULOCK**: Releases the previously placed lock on the part composed of **size** contiguous bytes pointed to by the file pointer.
- **F_LOCK**: Places a lock on the free part composed of **size** contiguous bytes pointed to by the file pointer. If the part to be locked is not free, the requesting process is blocked until the part becomes free. The process is not blocked if the system detects a deadlock.
- **F_TLOCK**: Places a lock on the free part composed of **size** contiguous bytes pointed to by the file pointer. If the part to be locked is not free, the **lockf()** system call returns -1 (lock attempt failure).
- **F_TEST**: Tests whether the part composed of **size** contiguous bytes pointed to by the file pointer is free or not. If the part to be locked is not free, **lockf()** returns -1.

"partage.c" demonstrates how to share a file without locking:

Code III.4 partage.c

```
#include <unistd.h> // Header file for Unix system calls

int main(void) {
   int i, fd = 1; // File descriptor for standard output
   if (fork()) { // Parent process
      for (i = 0; i < 5; i++) {
         write(fd, "parent writes\n", 14); // Write message to standard
output
         sleep(1); // Sleep for 1 second
      }
   }
```

```
else { // Child process
        for (i = 0; i < 4; i++) {
            write(fd, "child writes\n", 13); // Write message to standard
output
            sleep(1); // Sleep for 1 second
        }
    }
    return 0; // Exit the program
}
```

Exécution of the program partage.c :

pascal> gcc -o partage partage.c
pascal> partage
parent writes
child writes
parent writes
child writes
child writes
parent writes
parent writes
parent writes
parent writes
pascal>

Below is the content of the "verrou.c" program, which demonstrates file sharing with locking:

Code III.5 verrou.c

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(void) {
    int i, fd = 1;
    if (fork()) { // Parent process
        lseek(fd, 0, 0);
        if (lockf(fd, F_LOCK, 1) < 0) {
            write(fd, "parent lockf failed ", 20);
            return (-1);
        }
        for (i = 0; i < 5; i++) {
            write(fd, "parent writes\n", 14);
            sleep(1);
        }
```

```
        write(fd, "parent will release the lock\n", 30);
        lseek(fd, 0, 0);
        lockf(fd, F_ULOCK, 0);
        wait(NULL);
    } else { // Child process
        lseek(fd, 0, 0); // Lock byte 0
        if (lockf(fd, F_LOCK, 1) < 0) {
            write(fd, "child lockf failed\n", 19);
            return (1);
        }
        for (i = 0; i < 4; i++) {
            write(fd, "child writes\n", 13);
            sleep(1);
        }
        write(fd, "child will release the lock\n", 29);
        lseek(fd, 0, 0);
        lockf(fd, F_ULOCK, 0);
    }

    close(fd);
    return 0;
}
```

This program demonstrates file locking using lockf() function. The parent process writes to the file, locks it, writes again, and then releases the lock. Meanwhile, the child process also writes to the file, locks it, writes again, and then releases the lock. The locking mechanism ensures that their writes are not interleaved.

Execution of verrou.c:

pascal> gcc -o verrou verrou.c
pascal> verrou
parent writes
parent writes
parent writes
parent writes
parent writes
parent will release the lock
child writes
child writes
child writes
child writes
child will release the lock

pascal>

The output demonstrates the interleaved writes from both the parent and child processes, followed by the messages indicating the release of the lock by each process.

fcntl()

The fcntl() system call is actually much more complex because its prototype is: int fcntl(int fd, int command, ...); and it serves various purposes. Among them are duplicating descriptors, accessing descriptor attributes, and file locking.

#include <fcntl.h>
*int fcntl(int fd, int cmd, struct flock *lock);*

Here, **fd** is the file descriptor of the file to be locked. **lock** is a structure with the following fields:

- **l_type**: int Type of locking.
- **l_whence**: int equivalent to the lseek() system call with SEEK_SET, SEEK_CUR, and SEEK_END.
- **l_start**: Start of the locked portion of the file.
- **l_len**: Length locked in bytes.
- **l_pid**: Process ID. Provided by the system.

The type of locking can be:

- **F_RDLCK**: Shared lock for reading.
- **F_WRLCK**: Exclusive lock for writing.
- **F_UNLCK**: No lock.

cmd is one of the following operations:

- **F_GETLK**: Access the characteristics of an existing lock.
- **F_SETLK**: Setting or modifying a lock in non-blocking mode.
- **F_SETLKW**: Setting or modifying a lock in blocking mode.

The program lock.c locks a file until you press <Return> to unlock it.

Code III.6 lock.c

```
# include <fcntl.h>
# include <stdio.h>
# include <unistd.h>
# include <string.h>

int main (int argc, char argv[])
{
   char *file = argv[1];
   int fd;
   struct flock lock;
   printf("Opening %s\n", file);
   fd = open(file, O_RDWR);
```

```
    printf("Locking\n");
    memset(&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;
    fcntl(fd, F_SETLKW, &lock);
    printf("Locked. Press ENTER to unlock...");
    getchar();
    printf("Unlocking\n");
    lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLKW, &lock);
    close(fd);
    return 0;
}
```

In a terminal, create a TEST file and lock it with lock.c:

l*eibnitz> cat > TEST*
HOLA!
^D
leibnitz> ./lock TEST
Opening TEST
Locking
Locked. Press ENTER to unlock...

In another terminal, try to lock the same TEST file:

leibnitz> ./lock TEST
Opening TEST
Locking
The execution remains frozen until returning to the first terminal and pressing <Return>:
...
Locked. Press ENTER to unlock...
Unlocking

III.5.3 Locking with flock()

The Linux and Unix BSD systems provide flock(), another system call for file locking. This call is not standardized by Posix, but it is still in use.

#include <sys/file.h>
int flock(int fd, int operation);

Here, fd is the file descriptor, and it accepts the following operations:

LOCK_SH: Shared lock. Multiple processes can lock simultaneously.
LOCK_EX: Exclusive lock.
LOCK_UN: Unlock the fd.

LOCK_NB: Non-blocking lock.

Consider the following example that uses *flock* in the program flock.c

Code III.7 flock.c

```
# include <stdio.h>
# include <stdlib.h>
# include <sys/file.h>

void display(int fd)
{
   int data;
   flock(fd, LOCK_SH);
   while (read(fd, &data, sizeof(int)) > 0)
      printf("data: %d\n", data);
   close(fd);
}
void add(int fd)
{
   int data;
   flock(fd, LOCK_EX);
   do
   {
      printf("Data? ");
      scanf("%d", &data);
      write(fd, &data, sizeof(int));
   } while (data != 1);
   close(fd);
}
```

In one shell screen, execute **flock.c**:

pascal> flock

flock 1=Show 2=Add
pascal> flock 2
exclusive lock on fd=3
Data? 1
Data? 2
Data? 3
Data? 4

Without finishing, in a second shell, execute flock 1:

pascal> flock 1
shared lock on fd=3

Another process has the lock. Waiting

And in a third shell, execute **flock 2**:

pascal> flock 2
exclusive lock on fd=3
Another process has the lock. Waiting

If we return to the first session and finish adding data:

...
Data? 4
Data? -1
pascal>

Probably the third (or second) session will be unblocked, and we will have the data request. If we continue and finish entering data:

...
Another process has the lock. Waiting
Data? 5
Data? 6
Data? -1
pascal>

This is when the second (or third) session wakes up and shows its execution. Of course, we hope it will show the sequence 1, 2, 3, 4, 5, 6, -1, but what it actually shows is totally different:

...
Shared lock on fd=3 Another process has the lock. Waiting
Data: 5
Data: 6
Data: -1
Data: 4
Data: -1
pascal>

III.6 Buffer Cache

Processes cannot directly manipulate disk data. They must be brought into main memory. Many filesystems seek to reduce the number of disk accesses because the average disk access time is a few tens of milliseconds. The most common technique to reduce disk accesses is to use a buffer cache. The buffer cache is a space located in main memory where a set of disk blocks is loaded. Each time a file element is accessed, the desired block (the one containing the file element) is first examined to see if it is in the buffer cache. If it is, the request is fulfilled without having to access the disk. Otherwise, to fulfill the request, the

desired block must first be loaded from the disk into the buffer cache. If a block needs to be loaded and the buffer cache is full, one of the blocks in the buffer cache must be evicted to make room for the requested block. The evicted block must be copied back to the disk if it has been modified since loading. To choose the block to evict, one can use one of the page replacement algorithms (FIFO, LRU, etc.).

Problem: If blocks placed in the buffer cache have been modified and a failure occurs before they are copied to the disk, the filesystem enters an inconsistent state. To minimize the risk of inconsistency, the Unix system immediately copies blocks containing inodes and directories after their modification. Ordinary data blocks are manually copied if they need to be removed from the buffer cache or automatically by a daemon every 30 seconds. This daemon can also be triggered by the sync() system call. In MS-DOS, blocks in the buffer cache are copied to disk every time they are modified. The buffer cache is not the only way to improve filesystem performance. Another technique is to reduce the time it takes to search for a block on the disk.

III.7 Filesystem Consistency

Most computers have a utility program that checks filesystem consistency. This program may be executed at each system startup, especially following a forced shutdown. Filesystem consistency checking can be performed at two levels: block or file.

III.7.1 Block Level

At the block level, the checker constructs two tables. The first indicates, for each occupied block, the number of times the block is referenced in file inodes. The second indicates, for each free block, the number of times it appears in the list of free blocks (or the free block bitmap). If the filesystem is consistent, each block has a 1 either in the first table or in the second. If a block does not appear in either the first or second table, the block is said to be missing. The checker adds the missing blocks to the list of free blocks. If the block appears twice in the list of free blocks, the checker reconstructs the list of free blocks. The worst that can happen is that a block belongs to two or more files. If one of the two files is destroyed, the block will be placed in the list of free blocks (making it both free and used). If both files are destroyed, the block will appear twice in the list of free blocks. In this case, the best solution is to allocate a free block, copy the common block to it, and replace the common block in one of the two files with the new one. The last possibility of inconsistency in the two tables is that a block is both used and free. The solution in this case is to remove the block from the list of free blocks.

III.7.2 File Level

The checker checks consistency from the perspective of links. Each index node number should appear as many times in the tree structure as it has links. It starts at the root directory and recursively traverses the entire tree to determine, for each inode number, the number of references. It then compares the number of references obtained for each inode number with that contained in the corresponding inode. If the two numbers are different, the system is in an inconsistent state. The solution is to correct the number of references in the inode. In most Unix systems, the fsck program performs this task at each startup, if necessary. See the fsck manual page for more details.

III.8 Protection

Filesystems sometimes contain very important information. Therefore, they must protect this information against unauthorized access and loss. Causes of loss include natural disasters, hardware or software errors, and human errors. Most of these issues can be resolved by making regular backups. The problem of intruders is much more complex. To ensure protection against intruders, several mechanisms have been implemented, including user identification (passwords or physical identification) and object protection codes.

Chapter IV

PROTECTION

IV.1.Introduction

In an operating system, it's imperative to safeguard processes from interference with one another. This necessitates employing various mechanisms to ensure that only authorized processes sanctioned by the operating system can access and manipulate files, memory segments, CPU, and other system resources.

Protection entails a system for regulating the access of programs, processes, or users to the resources defined by a computer system. This system must offer a means to specify the controls to be applied, along with a mechanism for enforcement. It's important to distinguish protection from security, which pertains to the confidence level in preserving the integrity of a system and its data. Security assurance encompasses a broader spectrum than protection and will be further discussed in this chapter.

IV.2 Objectives of Protection

With the increasing complexity and ubiquity of computer systems, ensuring their integrity has become paramount. Initially conceived as a supplement to multiprogramming operating systems, protection aimed to enable secure sharing of common logical and physical resources among untrustworthy users. Modern protection concepts have evolved to enhance the reliability of any system employing shared resources.

Protection serves several purposes. Primarily, it aims to prevent intentional breaches of access restrictions by users. More significantly, it ensures that each active program component utilizes system resources in alignment with stated policies, crucial for the reliability of a system. Additionally, protection mechanisms can enhance reliability by identifying interface errors between subsystems early on, preventing the contamination of healthy subsystems by malfunctioning ones.

An unprotected resource is vulnerable to unauthorized or incompetent use. A protection-oriented system distinguishes between authorized and unauthorized usage, providing mechanisms for enforcement.

The role of protection in a computer system is to enforce policies governing resource utilization, which can be established in various ways. Some policies are inherent in system design, while others are defined by system management or individual users. Therefore, a protection system must offer flexibility to enforce diverse policies that may vary by application and evolve over time. Application programmers also need to utilize protection mechanisms to safeguard resources created and supported by application subsystems.

It's essential to differentiate between mechanisms and policies: mechanisms determine how something is done, while policies dictate what will be done. Separating policy from mechanism enhances flexibility, as policies may change over time, necessitating minimal alterations in underlying mechanisms.

IV.3 Protection Principles

A fundamental guiding principle in system design, particularly for operating systems, is the principle of least privilege. This principle advocates granting programs, users, and systems only the necessary privileges to execute their tasks effectively. An analogy can be drawn with a security guard possessing a passkey. If the passkey grants access solely to the public areas under the guard's purview, any misuse or compromise of the key would result in minimal damage. Conversely, if the passkey provides access to all areas, its loss, theft, misuse, or duplication would pose significant risks. An operating system adhering to the principle of least privilege ensures that its features, programs, system calls, and data structures are designed to minimize damage in case of failure or compromise. For instance, if a buffer overflow occurs in a system daemon, it should lead to the daemon's failure but not allow the execution of code from the process's stack, which could enable a remote user to gain maximum privileges and access the entire system. Such an operating system also offers system calls and services that enable applications to implement fine-grained access controls. It provides mechanisms to grant privileges when necessary and revoke them when not needed. Additionally, it establishes audit trails for all privileged function access, enabling tracking of protection and security activities by programmers, system administrators, or law enforcement officers. Implementing the principle of least privilege in user management involves creating separate accounts for each user with only the privileges required for their tasks. For example, an operator responsible for tape mounting and file backup has access only to the relevant

commands and files. Some systems employ role-based access control (RBAC) to facilitate this.

In a computing facility governed by the principle of least privilege, computers can be restricted to running specific services, accessing particular remote hosts via specific services, and doing so only during specific times. These restrictions are typically enforced through service enabling or disabling and access control lists. While the principle of least privilege aims to create a more secure computing environment, its effectiveness can vary. For instance, Windows 2000 has a complex protection scheme but still exhibits many security vulnerabilities. In contrast, Solaris, despite being a UNIX variant historically designed with minimal protection, is considered relatively secure. The differences may arise from factors such as the complexity of the operating system, the completeness of its protection scheme, and the focus of protection on the right aspects of the system.

IV.4 Domain of Protection

A computer system comprises processes and objects, including both hardware (e.g., CPU, memory segments, printers, disks, tape drives) and software (e.g., files, programs, semaphores) objects. Each object has a unique name and can be accessed only through well-defined operations. The operations possible for an object depend on its type. For instance, a CPU can only be executed, memory segments can be read and written, while a CD-ROM or DVD-ROM can only be read. Processes should only access resources for which they have authorization and should be limited to accessing resources necessary for their current task, following the need-to-know principle. This principle restricts access to variables and files to only those relevant to the task, minimizing the risks of security violations.

IV.4.1 Domain Structure

In the context of protection mechanisms, processes operate within defined protection domains, each outlining the permissible access to resources. Each domain delineates a specific set of objects and the corresponding operations allowed on these objects. The capability to execute an operation on an object is represented as an access right. A domain comprises a collection of such access rights, typically denoted as ordered pairs <object name, rights set>. For instance, if Domain 0 possesses the access right <file P, {read, write}>, processes operating within Domain 0 can both read from and write to file P, but are restricted from executing any other operation on that particular object.

Domains need not be mutually exclusive; they can share access rights. For example, as illustrated in FIG IV.1, consider three domains: D1, D2, and D3. The access right <object O4,

{print}> is shared between D2 and D3, implying that processes running in either of these domains can execute the print operation on object O4. Conversely, only processes within domain D1 may execute operations on object O3. The linkage between a process and a domain can be either static or dynamic. In static associations, the set of resources accessible to a process remains constant throughout its lifespan, while dynamic associations allow for variations. Establishing dynamic protection domains is inherently more complex compared to static ones. In cases where processes are statically associated with domains, and adherence to the need-to-know principle is desired, mechanisms must be in place to alter domain contents. This necessity arises from scenarios where a process operates in multiple phases, each requiring different access rights. For instance, a process may necessitate read access in one phase and write access in another. If a domain is static and encompasses both read and write access, it provides more privileges than required in each phase, leading to inefficiencies.

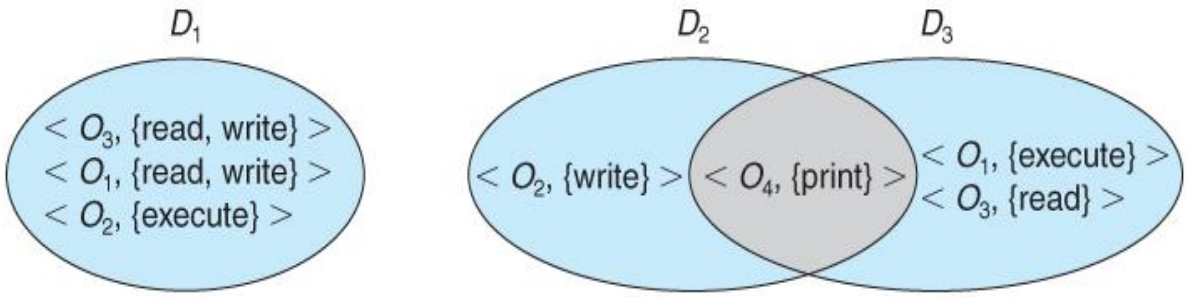


FIG IV.1 System with three protection domains

VI.4.2 Domain Realization

To uphold the need-to-know principle, it is imperative to allow the modification of domain contents to consistently reflect the minimum necessary access rights. When associations between processes and domains are dynamic, mechanisms enabling domain switching are essential, empowering processes to transition between domains. This flexibility is crucial as it facilitates the adjustment of domain content when necessary. If static associations prohibit domain content alteration, the same outcome can be achieved by creating new domains with adjusted contents and switching to these new domains as required.

Domains can be implemented in various ways:

- Each user can function as a domain, with the set of accessible objects contingent upon the user's identity. Domain switching occurs during user changes, typically when one user logs out and another logs in.

- Each process can serve as a domain, with the set of accessible objects determined by the process's identity. Domain switching occurs when a process sends a message to another process and awaits a response.
- Each procedure can act as a domain, with the accessible objects corresponding to the local variables defined within the procedure. Domain switching occurs upon procedure invocation.

Further exploration of domain switching is provided in the folowing sections.

In the standard dual-mode (monitor-user mode) model of operating-system execution, processes executing in monitor mode have the privilege to execute privileged instructions, thereby gaining complete control of the computer system. Conversely, processes executing in user mode are restricted to invoking only nonprivileged instructions, operating within predefined memory spaces. These modes serve to shield the operating system (executing in the monitor domain) from user processes (executing in the user domain). However, in a multiprogrammed operating system, a more sophisticated scheme is necessary to ensure users are protected from each other. This requirement prompts the development of an elaborate protection mechanism, as exemplified by examining two influential operating systems: UNIX and MULTICS.

IV.4.3 Example: UNIX

In UNIX, a domain is associated with the user, and switching domains equates to temporarily altering user identification. This domain switch is facilitated through the file system: each file is associated with an owner identification and a domain bit (setuid bit). When the setuid bit is set, and a user executes the file, their user ID is temporarily changed to that of the file's owner. Conversely, when the setuid bit is unset, the user ID remains unchanged. For instance, when user A executes a file owned by B with the setuid bit off, the user ID remains as A. However, if the setuid bit is on, the user ID changes to that of the file owner (B) during execution. This temporary user ID alteration ends when the process exits. Various methods are employed in operating systems where user IDs serve as the basis for defining domains, as most systems require such a mechanism to provide privileged facilities to general users. This mechanism becomes necessary when granting access to privileged functions without allowing users to develop their own programs for these tasks. For instance, on UNIX systems, setting the setuid bit on a networking program enables the user ID to change when the program is executed. The user ID is altered to that of a user with network access privilege, such as root, the most powerful user (UID 0). However, a drawback of this

method is that if a user manages to create a file with UID root and with its setuid bit on, they can exploit it to gain root access and execute commands with unrestricted privileges. The setuid mechanism is further elaborated in Appendix A.

An alternative approach, adopted by some operating systems, involves placing privileged programs in a designated directory. The operating system is designed to adjust the user ID of any program executed from this directory, either to that of root or to the owner of the directory. This mitigates some security risks associated with setuid programs, where attackers may create and conceal them with obscure file or directory names for later exploitation. However, this method is less versatile compared to the approach used in UNIX. More stringent systems simply disallow any alteration of the user ID, necessitating special techniques to grant users access to privileged facilities. For instance, a daemon process may be initiated during boot time and run with special user privileges. Users then interact with a separate program, which sends requests to this daemon whenever they require access to the privileged function. This methodology is exemplified by the TOPS-20 operating system. In all these systems, meticulous attention must be paid when developing privileged programs. Any oversight can lead to a complete absence of protection on the system, making these programs prime targets for attackers attempting to infiltrate the system. Unfortunately, attackers often succeed in exploiting security vulnerabilities, as seen in numerous breaches on UNIX systems due to the setuid feature.

IV.4.4 Example: MULTICS

In the MULTICS operating system, protection domains are structured hierarchically into a ring configuration. Each ring corresponds to a distinct domain, with rings numbered from 0 to 7 (refer to FIG IV.2). If $i < j$, then the domain represented by ring j is a subset of the domain represented by ring i, implying that processes executing in lower-numbered rings have more privileges than those in higher-numbered rings. The highest privilege level is attributed to processes executing in domain D0. In systems with only two rings, this setup mirrors the monitor-user mode of execution, where monitor mode corresponds to ring 0 and user mode to ring 1. MULTICS employs a segmented address space, where each segment corresponds to a file and is associated with one of the rings. A segment description includes an entry identifying the ring number, along with three access bits controlling reading, writing, and execution permissions. The mapping between segments and rings is a policy decision beyond the scope of our discussion here.

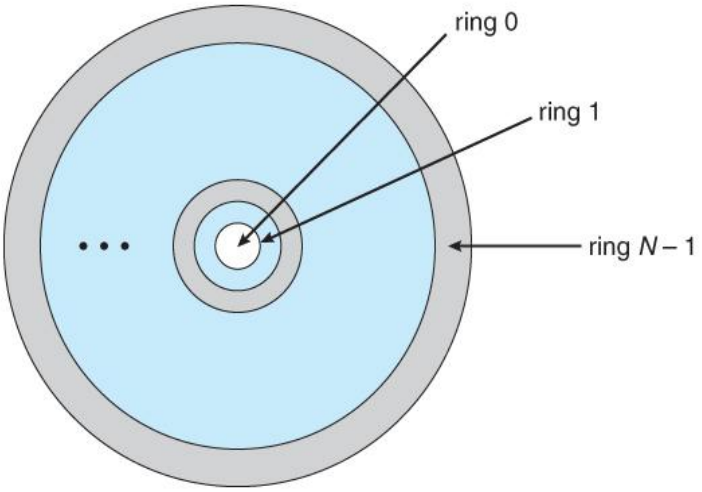


FIG IV.2 MULTICS ring structure

Each process in MULTICS is associated with a current-ring-number counter, indicating the ring in which the process is currently executing. When a process operates in ring i, it cannot access a segment linked with ring j (where $j < i$), but it can access a segment associated with a lower or equal ring number (i.e., ring k, where $k \leq i$). However, the type of access is restricted based on the access bits assigned to that segment. Domain switching in MULTICS happens when a process transitions from one ring to another by invoking a procedure in a different ring. This transition must be controlled to prevent unauthorized access. To facilitate controlled domain switching, the ring field of the segment descriptor is modified to include:

- Access bracket: A pair of integers (b1, b2) where $b1 \leq b2$.
- Limit: An integer b3 where $b3 > b2$.
- List of gates: Identifies the entry points (or gates) through which the segments can be invoked.

If a process executing in ring i calls a procedure with an access bracket (b1, b2), the call is permitted if $b1 \leq i \leq b2$, and the process remains in ring i. Otherwise, a trap to the operating system occurs, and the situation is managed as follows:

- If $i < b1$, the call is allowed because it represents a transition to a ring with fewer privileges. However, if the call involves passing parameters referring to segments in a lower ring, these segments must be duplicated into an accessible area for the called procedure.
- If $i > b2$, the call is permitted only if b3 is greater than or equal to i, and the call targets one of the designated entry points in the list of gates. This mechanism enables

processes with restricted access rights to invoke procedures in lower rings but in a controlled manner.

One drawback of the ring structure is its inability to enforce the need-to-know principle. Specifically, if an object should be accessible in domain i but not in domain j, then i < j. However, this requirement implies that every segment accessible in domain j is also accessible in domain i.

Although the protection system in MULTICS is more intricate and less efficient compared to modern operating systems, its complexity is justified in scenarios where it's crucial to balance protection with system usability and performance. For instance, a complex protection system is suitable for a university computer handling sensitive data like student grades, whereas a simpler protection system may suffice for a high-performance computing environment where efficiency is paramount. Separating mechanism from policy allows for flexibility, enabling the same system to adopt complex or simple protection measures based on user requirements. To achieve this separation, a more generalized protection model is necessary.

IV.5 Access Matrix

Our protection model can be abstractly represented as a matrix, known as an access matrix. In this matrix, the rows correspond to domains, and the columns represent objects. Each entry in the matrix contains a set of access rights indicating what operations a process in a particular domain can perform on a specific object. Since the objects are explicitly defined by the columns, the object names can be omitted from the access rights. To illustrate this concept, let's examine the access matrix depicted in Table IV.1. In this example, there are four domains and four objects: three files (F1, F2, F3) and one laser printer. For instance, a process operating in domain D1 has the privilege to read files F1 and F3. Conversely, a process in domain D4 enjoys the same privileges as one in domain D1, but additionally, it can write to files F1 and F3. It's important to note that access to the laser printer is restricted to processes executing in domain D2.

The access matrix scheme offers a mechanism for defining various policies concerning protection. This mechanism involves the implementation of the access matrix itself and ensuring that the specified semantic properties are upheld. Specifically, it ensures that a process operating in a particular domain can only access objects listed in its row and then only as permitted by the entries in the access matrix.

Table IV.1 Access matrix

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Policy decisions related to protection are determined by the contents of the access matrix entries. These decisions include defining which rights should be granted in each entry. Additionally, the policy governing the domain in which each process operates is typically established by the operating system. Users play a role in defining the access matrix entries. When a user creates a new object, its corresponding column is added to the access matrix with the appropriate access rights, as determined by the creator. Users have the flexibility to specify different rights for different entries in the column, as needed. The access matrix serves as an effective mechanism for enforcing strict control over both static and dynamic associations between processes and domains. When a process switches from one domain to another, it effectively performs an operation on the domain object itself. Similarly, modifying the content of the access matrix is akin to operating on the access matrix object. By including domains and the access matrix as objects in the access matrix, we can control these operations.

Considering each entry in the access matrix as an object to be protected, we then determine the operations permitted on these new objects (domains and the access matrix) and how processes are allowed to execute these operations. Processes should have the capability to switch from one domain to another. Domain switching from domain Di to domain Dj is permissible if and only if the access right "switch" is granted, denoted by access(i, j). As depicted in Table IV.2, a process in domain D2 can switch to either domain D3 or D4. Similarly, a process in D4 can switch to D1, and one in D1 can switch to D2.

Table IV.2 Access matrix of Table IV.1 with domains as objects

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Tables IV.3 Access matrix with *copy* rights

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Enabling controlled modifications to the contents of access-matrix entries necessitates the introduction of three additional operations: copy, owner, and control. Let's delve into each of these operations.

1. **Copy Operation:** The copy operation permits the duplication of an access right from one domain (or row) of the access matrix to another. This operation is indicated by appending an asterisk (*) to the access right. Importantly, the copy right allows the replication of the access right only within the same column (i.e., for the specific object) where the right is defined. For instance, in Table IV.3 (a), a process in domain

D2 can copy the read operation into any entry associated with file F2. This scheme presents two variants:

- Transfer of a right: A right is moved from access(i, j) to access(k, j), subsequently being removed from access(i, j). This action is characterized as a transfer rather than a copy.
- Limited propagation: When the right R* is copied from access(i, j) to access(k, j), only the right R (excluding other rights like W) is created. This restricts further copying of the right R by a process executing in domain Dk. A system may opt for one of these copy rights, or it may offer all three as distinct rights: copy, transfer, and limited copy.

2. **Owner Operation:** The owner right governs the addition and removal of rights within access-matrix entries. If access(i, j) includes the owner right, a process in domain Di possesses the authority to add or remove any right within any entry in column j. For example, in Tables IV.4(a), domain D1 serves as the owner of F1, granting it the capability to add or delete any valid right in column F1.

These operations provide mechanisms for managing access rights within the access matrix, ensuring controlled modifications to access permissions as needed by processes operating within different domains.

Similarly, domain O2 assumes ownership of F2 and F3, granting it the authority to add or remove any valid rights within these columns. Consequently, the access matrix depicted in Tables IV.4(a) can be altered to reflect the modifications shown in Tables IV.4(b). While the copy and owner rights enable processes to manipulate entries within columns, there's also a need for a mechanism to modify entries within rows. This role is fulfilled by the control right, which exclusively applies to domain objects. If access(i, j) includes the control right, a process executing in domain OJ possesses the capability to eliminate any access right from row j. For instance, if the control right is included in access(02, F04), then a process operating in domain O2 could modify domain 0 4 accordingly, as demonstrated in Table IV.5. Although the copy and owner rights offer a means to restrict the propagation of access rights, they lack the capability to prevent the dissemination or disclosure of information. This challenge, known as the confinement problem, is generally unsolvable.

Tables IV.4 Access matrix with *owner* rights

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

While these operations on domains and the access matrix may seem trivial, they highlight the access-matrix model's capacity to implement and manage dynamic protection requirements. The model permits the dynamic creation of new objects and domains, showcasing its flexibility. However, it's crucial to recognize that we've merely introduced the fundamental mechanisms here; the decisions regarding which domains should access which objects and in what manner must be determined by system designers and users.

Table IV.5 Modified access matrix of Table IV.4

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

IV.6 Implementation of Access Matrix

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although data structure techniques are available for representing sparse matrices, they are not particularly useful for this application because of the way in which the protection facility is used. Here, we first describe several methods of implementing the access matrix and then compare the methods.

IV.6.1 Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <domain, object, rights-set>. Whenever an operation M is executed on an object 0i within domain n, the global table is searched for a triple <D;, ai, Rk>, with M ∈ Rk. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised. This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept entirely in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, it must have a separate entry in every domain.

IV.6.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object as described previously. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs <domain, rights-set>, which define all domains with a nonempty set of access rights for that object. This approach can be extended easily to define a list plus a default set of access rights. When an operation M on an object 0i is attempted in domain Oi, we search the access list for object 0i, looking for an entry <n, R> with M ∈ R. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

IV.7 Implementation Strategies for Access Matrix

How can we effectively implement the access matrix? Typically, the matrix will exhibit sparsity, meaning that most entries will remain empty. Although techniques exist for handling sparse matrices in data structures, they might not be the most suitable for this

particular application due to the unique characteristics of the protection mechanism. Let's explore several methods for implementing the access matrix and then compare their efficacy.

IV.7.1 Global Table Approach

A straightforward way to implement the access matrix is by using a global table composed of ordered triplets <domain, object, rights-set>. Whenever an operation, denoted as M, is performed on an object Oi within domain n, the global table is queried to find a matching triplet <Di, Oi, Rk>, where M is a member of Rk. If such a triplet is found, the operation proceeds; otherwise, an exception or error is raised. However, this method has its limitations. The table tends to be large and may not entirely fit into main memory, necessitating additional input/output operations. Often, virtual memory techniques are employed to manage this table. Moreover, it's challenging to leverage any special groupings of objects or domains. For example, if all users require read access to a specific object, it must have a separate entry in each domain.

IV.7.2 Access Lists for Individual Objects

Alternatively, each column in the access matrix can be represented as an access list for a single object. Empty entries can be discarded to reduce redundancy. The resultant list for each object comprises ordered pairs <domain, rights-set>, defining all domains with a non-empty set of access rights for that object. This approach can be further extended to include a default set of access rights. When attempting an operation M on an object Oi in domain Di, we first search the access list for Oi, looking for an entry <n, R> where M is a member of R. If found, the operation proceeds; otherwise, we check the default set. If M is in the default set, access is granted; otherwise, access is denied, triggering an exception condition. For efficiency, it may be prudent to check the default set before searching the access list.

IV.7.3 Capability Lists for Domains

Instead of linking columns of the access matrix with objects via access lists, another approach associates each row with its respective domain. Here, a capability list for a domain comprises a roster of objects along with the permissible operations on those objects. An object is often denoted by its physical name or address, referred to as a capability. When executing operation M on object Oi, the process employs the operation M and provides the capability (or pointer) for Oi as a parameter. Possession of the capability automatically grants access.

The capability list, while affiliated with a domain, remains inaccessible directly to a process executing within that domain. It is safeguarded as a protected object, overseen by the

operating system and only indirectly accessed by users. This protection method hinges on ensuring that capabilities never infiltrate any user-accessible address space where they could be tampered with. If all capabilities are secure, the objects they guard remain protected against unauthorized access. Capabilities were originally proposed as secure pointers to address the growing need for resource protection as multiprogrammed computer systems evolved. The concept of inherently secure pointers lays the groundwork for extending protection up to the application level.

For effective inherent protection, capabilities must be discernible from other types of objects and interpreted by an abstract machine that higher-level programs run on. Typically, capabilities are distinguished from other data in one of two ways:

1. Each object carries a tag indicating its type, either as a capability or accessible data. These tags remain inaccessible to application programs directly, with hardware or firmware support potentially enforcing this restriction. While a single bit suffices to differentiate between capabilities and other objects, additional bits are often employed for tagging all objects by their types at the hardware level. This approach allows hardware to distinguish between integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values based on their tags.
2. Alternatively, a program's address space can be divided into two segments. One segment, accessible to the program, contains the program's regular data and instructions, while the other segment housing the capability list is exclusively accessible by the operating system. Utilizing a segmented memory space proves advantageous in supporting this approach.

Numerous capability-based protection systems have been developed, briefly outlined in Section IV.10. Additionally, the Mach operating system employs a variant of capability-based protection, further elaborated in Appendix B.

IV.7.4 A Lock-Key Mechanism

The lock-key scheme represents a middle ground between access lists and capability lists. In this system, each object maintains a list of unique bit patterns known as locks, while each domain holds a list of unique bit patterns termed keys. Access to an object by a process executing within a domain is only permissible if the domain possesses a key matching one of the locks associated with the object. As with capability lists, managing the list of keys for a

domain falls under the purview of the operating system on behalf of that domain. Users lack the privilege to directly inspect or alter the list of keys (or locks).

IV.7.5 Comparison

Let's compare the various techniques for implementing an access matrix:

- **Global Table:** Using a global table is straightforward, but it can become unwieldy due to its potentially large size. It often fails to leverage special groupings of objects or domains.
- **Access Lists:** Access lists align well with user needs as they allow users to specify access permissions for objects and domains. However, determining the set of access rights for each domain can be challenging since the access-rights information is not localized. Additionally, every access to the object requires a search of the access list, which can be time-consuming in large systems.
- **Capability Lists:** While not directly aligned with user needs, capability lists excel in localizing information for a given process. A process seeking access must present a capability, simplifying the protection system's task of verifying its validity. However, revoking capabilities may be inefficient.
- **Lock-Key Mechanism:** This mechanism serves as a compromise between access lists and capability lists. Its effectiveness and flexibility depend on the length of the keys. Keys can be freely passed from domain to domain, and access privileges can be efficiently revoked by adjusting the locks associated with the object.

Most systems employ a combination of access lists and capabilities. Initially, when a process attempts to access an object, the access list is searched. If access is granted, a capability is created and attached to the process for swift subsequent access. After the last access, the capability is revoked. This strategy is exemplified in systems like MULTICS and CAL. For instance, in a file system with associated access lists, when a process opens a file, the directory structure is searched, access permission is checked, and buffers are allocated. This information is recorded in a file table associated with the process. Subsequent operations on the file are facilitated through references to this file table. When the file is closed, the file-table entry is deleted, ensuring controlled access.

Despite the capability in the file table entry, access rights must still be verified on each access attempt. If a file is opened for reading, only read access capability is granted in the file-table

entry. Any attempt to write to the file is flagged as a protection violation by comparing the requested operation with the capability in the file-table entry.

IV.8 Access Control

Previously, we explored access controls for files within a file system, where each file and directory are assigned ownership, group, or user-based access-control information. Similar access control mechanisms can be extended to other aspects of a computer system, exemplified by Solaris 10. Solaris 10 introduces advancements in protection within the Sun Microsystems operating system by incorporating the principle of least privilege through role-based access control (RBAC). RBAC revolves around privileges, which represent the rights to execute system calls or utilize specific options within those calls, such as opening a file with write access. Processes can be assigned privileges, limiting them to only the necessary access required for their tasks. Furthermore, privileges and programs can be assigned to roles. Users are then assigned these roles or can adopt them using role-based passwords. This approach enables users to assume roles granting specific privileges, allowing them to execute programs to accomplish particular tasks, as illustrated in FIG IV.3. By implementing privileges in this manner, the security risks associated with superusers and setuid programs are reduced.

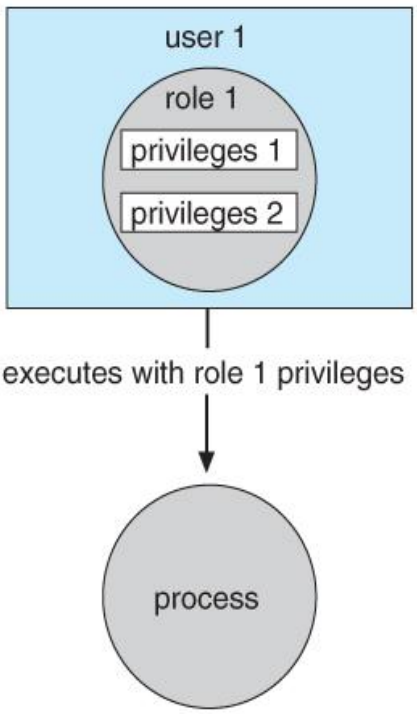


FIG IV.3 Role-based access control in Solaris 10

IV.9 Revocation of Access Rights

In dynamic protection systems, revoking access rights to shared objects among users can raise several questions regarding the nature and timing of revocation:

- **Immediate versus Delayed:** Does revocation happen instantly, or is there a delay? If delayed, can the timing of revocation be determined?
- **Selective versus General:** Does revocation affect all users with access rights to an object, or can it target specific users?
- **Partial versus Total:** Can only a subset of rights associated with an object be revoked, or must all access rights be revoked?
- **Temporary versus Permanent:** Can access be permanently revoked, or is there a possibility of regaining access later?

Revocation in an access-list system is straightforward. Access rights marked for revocation are simply deleted from the list. This process is immediate and can be applied generally or selectively, for total or partial access rights, and can be permanent or temporary.

However, revocation in capability-based systems poses a more complex challenge. Since capabilities are dispersed across the system, locating and revoking them requires specialized methods:

- **Reacquisition:** Capabilities are periodically deleted from each domain. If a process attempts to use a revoked capability, it discovers the deletion and may try to reacquire it. Access is denied if the capability has been revoked.
- **Back-Pointers:** Each object maintains a list of pointers to associated capabilities. During revocation, these pointers are followed to update or remove capabilities as needed. This approach, used in the MULTICS system, is effective but resource-intensive.
- **Indirection:** Capabilities indirectly point to objects via a global table. Revocation involves deleting the corresponding entry from this table. When an access is attempted, the capability leads to an invalid table entry, preventing access. This method, employed in the CAL system, does not support selective revocation.
- **Keys:** Each capability is associated with a unique key, set during its creation and compared with the master key associated with the object during execution. Revocation involves replacing the master key, rendering previous capabilities invalid. This method allows selective revocation when combined with a list of keys per object or a global table of keys.

In key-based schemes, operations like defining, inserting, and deleting keys should be restricted to authorized users, typically the object's owner. However, the decision on access to these operations is a policy choice left to the protection system's implementation rather than its definition.

IV.10 Capability-Based Systems

In this section, we examine two capability-based protection systems, each offering distinct features and policies. Although not widely adopted, these systems serve as experimental grounds for protection theories.

IV.10.1 Example: Hydra

Hydra, a capability-based protection system, offers significant flexibility by providing a predefined set of access rights interpreted by the system. Users can also define their rights, interpreted solely by their programs, with the system ensuring protection for both user-defined and system-defined rights. Object operations are procedurally defined, with procedures implementing these operations acting as objects themselves, accessed indirectly through capabilities. User-defined procedure names must be identified to Hydra for handling objects of user-defined types. When an object's definition is known, operation names become auxiliary rights, which must be included in a capability for a process to perform an operation on an object. Hydra features rights amplification, allowing certified procedures to act on specific types of formal parameters with rights independent of the calling process. However, this trust must not extend universally, and caution is needed to prevent unauthorized actions.
Rights amplification enables procedures to access representation variables of abstract data types indirectly. For instance, a process holding a capability for a typed object may invoke an operation, indirectly accessing the object's representation. This dynamic adjustment of rights ensures consistency in programmer-defined abstractions.

Hydra addresses mutual suspicion between subsystems by restricting amplifications. Its procedure-call mechanism tackles the risk of service programs malfunctioning or retaining unauthorized access rights to data. Subsystems built atop Hydra's protection kernel enforce policies on resource usage by user processes, with the system providing standard access protection through capabilities. Hydra empowers programmers with direct access to its protection system, offering a library of system-defined procedures callable by user programs. Users can incorporate these procedures into their code or use a program translator interfaced with Hydra. This section outlines the features and capabilities of Hydra, a capability-based

protection system, highlighting its flexibility and mechanisms for enforcing access control and rights amplification.

IV.11 Language-Based Protection

Protection in computer systems is traditionally handled by the operating system kernel, which verifies access attempts to protected resources. However, comprehensive access validation can incur significant overhead, necessitating either hardware support or compromises in protection goals. As operating systems evolve to offer higher-level user interfaces, protection systems increasingly draw from programming language concepts, particularly abstract data types and objects.

IV.11.1 Compiler-Based Enforcement

Programming languages play a crucial role in specifying access control requirements for shared resources. By integrating protection declarations with data typing, subsystem designers can articulate protection needs and resource usage requirements directly within their programs. Compiler-based enforcement offers several advantages:

1. **Simplicity:** Protection requirements are declared rather than programmed as a sequence of operating system procedure calls, facilitating clarity and ease of implementation.
2. **Independence:** Protection specifications are independent of the facilities provided by a specific operating system, offering flexibility in enforcing protection.
3. **Enforcement:** Enforcement mechanisms need not be provided by subsystem designers, freeing them from the burden of implementation.
4. **Natural Notation:** Declarative protection specifications align well with the linguistic concept of data types, enhancing readability and comprehension.

Programming languages can employ various techniques to enforce protection, although they rely on underlying hardware and operating system support to some extent. For instance, software capabilities provided by systems like CAP can enable enforcing protection policies specified in the language. Compiler-supported enforcement separates references where protection violations are unlikely from those where violations might occur, contributing to security. While enforcement by a protection kernel offers higher security, compiler-based enforcement provides flexibility and efficiency advantages. Language-based enforcement

allows static access validation at compile time and tailored enforcement mechanisms, minimizing overhead.

The integration of protection concepts into programming languages is still in its early stages but holds promise for system design. As data security requirements become more stringent, the importance of suitable language notations for expressing protection requirements will grow, making protection a central concern for system designers in the era of distributed architectures. This section discusses the role of programming languages in enforcing access control and protection in computer systems, highlighting the advantages of compiler-based enforcement and the growing importance of protection in system design.

IV.11.2 Protection in Java

Java was crafted to operate within distributed environments, embedding robust protection mechanisms within the Java Virtual Machine (JVM). In Java, programs are structured into classes, consisting of data fields and methods. The JVM dynamically loads classes upon requests to create class instances. One of Java's notable features is its ability to load untrusted classes over networks and execute them within the same JVM, emphasizing the criticality of protection measures. Stack inspection is illllstrated in FIG IV.9. Here, the gui () method of a class in the *untrustcd applet* protection domain perforrI1s two operations, first a get () and then an open ().

Protection Decisions within the JVM

Java's approach to protection decisions diverges from traditional methods by handling them internally within the JVM, rather than relying solely on the operating system. Upon loading a class, it is assigned to a protection domain based on its source URL and digital signatures. Permissions for the domain, and consequently its classes, are dictated by a configurable policy file. This ensures that classes from various sources are endowed with appropriate permissions.

Stack Inspection

Java employs stack inspection to determine responsibility for access requests to protected resources. Each thread within the JVM maintains a stack of ongoing method invocations. When a method executes an access request, it may do so within a **doPrivileged** block, thereby asserting privilege for the request. Stack inspection scrutinizes the calling thread's stack frames to ascertain whether the request should be granted based on annotations and permissions associated with the protection domain.

Enforcement of Type Safety

Java's stringent load-time and run-time checks enforce type safety, preventing classes from accessing memory arbitrarily. This safeguard ensures that classes cannot manipulate their runtime stack or other components of the protection system. Type safety also facilitates encapsulation and protection of data and methods within classes, enforcing access restrictions based on access modifiers such as **private** or **protected**. Java's protection mechanisms, encompassing protection domain assignment, stack inspection, and enforcement of type safety, serve as pillars for secure execution within the JVM. By handling protection decisions internally and rigorously enforcing type safety, Java establishes a robust security framework suitable for distributed environments.

Table IV.6 Stack inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission
			(a, connect);
			connect (a);
			...

Chapter V

NETWORKS UNDER LINUX

V.1 Introduction

Linux networking encompasses the tools and protocols used to establish communication between devices over networks. It includes managing network interfaces, routing traffic, configuring firewalls, running network services, sharing files, and monitoring network activity. With its robust networking stack and versatile command-line tools, Linux offers a flexible and powerful platform for networking tasks.

V.2 TCP/IP

V.2.1 Basics

The origins of TCP/IP stem from research conducted by DARPA (Defense Advanced Research Project Agency), which began in 1970 and resulted in ARPANET. In practice, DARPA funded the University of Berkeley, which integrated the basic TCP/IP protocols into its UNIX BSD 4 system. TCP/IP gained popularity due to its generic data exchange programming interface between network machines, known as socket primitives, and the integration of application protocols. The TCP/IP protocols are overseen by the IAB (Internet Activities Board), which in turn supervises two other organizations:

- The IRTF (Internet Research Task Force), responsible for protocol development.
- The IETF (Internet Engineering Task Force), responsible for the Internet network.

Network addresses are distributed by the NIC (Network Information Center). The entire TCP/IP protocol suite is described in RFC (Request For Comments) documents.

- The lower layer is IP (Internet Protocol).
- The transport layer consists of TCP (Transmission Control Protocol) or UDP (User Datagram Protocol).
- The upper layers include application protocol layers, such as:

- NFS (Network File System): remote file sharing.
- DNS (Domain Name System): host-IP association.
- FTP (File Transfer Protocol): file transfer.
- TELNET: text terminal emulation...

The represented IP protocol version is V4. The future, already present, is the IPV6 protocol. Compatible with IPV4, it offers addressing on 128 bits (16 bytes), allowing for the expansion of network capabilities, especially in terms of size and addressing.

V.2.2 Addressing

V.2.2.1 Classes

It is important to know before installation what type of network the new server should integrate into, TCP/IP of course, but it already needs to reserve an IP address, a hostname (network machine name), know various gateways, the domain name, the class used, and the subnet mask. Here is a brief reminder about IP classes. An IP address is defined on 32 bits and represented by four numbers separated by dots: n1.n2.n3.n4. This address consists of two parts that define the network address and the host within the network. Depending on the case, four or five classes of addresses can be distinguished: A, B, C, D, and E, but only the first three concern us.

Note in the following that the legend will be as follows: N and h are bits, N identifying the network and h identifying the machine.

Class A

0NNNNNNN hhhhhhhh hhhhhhhh hhhhhhhh

1.x.x.x to 126.x.x.x.

n1 is between 1 and 126. 16,777,214 hosts, 127 networks.

Class B

10NNNNNN NNNNNNNN hhhhhhhh hhhhhhhh

128.0.x.x to 191.255.x.x. n1 is between 128 and 191. 65,534 hosts, 16,382 networks.

Class C

11NNNNNN NNNNNNNN NNNNNNNN hhhhhhhh

192.0.0.x to 223.255.255.x.

n1 is between 192 and 223. 254 hosts, 2,097,150 networks.

Class D Begins with 1110, for IP multicast.

Class E Begins with 1111, for experimentation.

There are host addresses that cannot be used. For example, in class C, only 254 hosts can be used, while the machine identifier is encoded on 8 bits (thus 256 values). This is because address 0 represents the network address, and address 255 represents the broadcast (multicast) address. Note that the following addresses should not be routed on the Internet and are reserved for local networks.

10.0.0.0 - 10.255.255.255 (10/8)

172.16.0.0 - 172.31.255.255 (172.16/12)

192.168.0.0 - 192.168.255.255 (192.168/16).

The address 127.0.0.1 is the loopback or self-loop address: it represents the machine itself, as well as the 127.0.0.0/8 subnet.

V.2.2.2 Subnets

Furthermore, it is possible to divide these networks into subnets using masks for finer addressing. A netmask is a binary mask that immediately separates the network and subnet address from the host address in the global IP address. The predefined masks are:

Class A: 255.0.0.0

Class B: 255.255.0.0

Class C: 255.255.255.0 To communicate directly with each other, hosts must belong to the same network or subnet. Calculating a subnet is quite simple. Here's an example for a class C network.

Network: 192.168.1.0

Network address: 192.168.1.255

Network mask: 255.255.255.0

Calculate a subnet mask:

To calculate the subnet mask, first determine how many machines you want to integrate into it. A class C network allows for 254 machines (0 and 255 being reserved). You want to create networks containing 60 machines. Add 2 to this value for reserved addresses (subnet address and broadcast address), which gives 62. Once the number of machines is determined, find the exact power of two or just greater than the number found. 2 to the power of 6 gives 64.

Write the mask in binary, place all bits of the class C network mask to 1 and place 0 for the first 6 bits of the mask corresponding to the machine part: 11111111 11111111 11111111 11000000.

Convert this mask to decimal: 255.255.255.192, and calculate all possible subnets. Since you are in a class C network, you can still vary the last two bits of the machine part:

00xxxxxx: 255.255.255.0

01xxxxxx: 255.255.255.64

10xxxxxx: 255.255.255.128

11xxxxxx: 255.255.255.192

In the end, you get four subnets of 62 machines each, totaling 248 machines. You end up with 256 if you add the four broadcast addresses and the four network addresses.

V.2.2.3 Routing

The network mask determines whether a destination machine is on the same network as yours or not. It is necessary to indicate the path packets should take to reach their destination IP. If your machine is a client workstation with only one network card and this network has only one router (typical case of a connection to the Internet), then you must create two routes. The first one indicates which network interface the packets should use to access the rest of the network (or subnet), and the second one indicates which route the packets should take to exit the network. Generally, this is referred to as a default route when only one router is present.

- To network1: use left network interface.
- To network2: use right network interface.
- To others: use right network interface to router1. Example 1 Here is an example where we have a Class C Network1 192.168.1.0 on eth0, a Class B Network2 172.16.0.0 on eth1, with a router address 192.168.1.254. Network Mask Interface Gateway 192.168.1.0 255.255.255.0 eth0 eth0 172.16.0.0 255.255.0.0 eth1 eth1 0.0.0.0 0.0.0.0 eth0 192.168.1.254 All network packets to 192.168.1.0 will pass through eth0. All packets destined for 172.16.0.0 will pass through eth1. By default, all other packets for unspecified networks will pass through eth0 and will be handled by gateway 192.168.1.254, which will route the packets. 1.3 Configuration 1.3.1 Network

Interface Configuration The network interface configuration file under Debian is located in /etc/network/interfaces.

 # cat interfaces

auto lo eth0 eth1

iface lo inet loopback

iface eth0 inet static

address 192.161.1.60

netmask 255.255.255.0

broadcast 192.168.1.255

gateway 192.168.1.1

iface eth1 inet dhcp

This example shows three types of interfaces:

- The loopback interface lo,
- Interface eth1 using DHCP, requiring no further configuration,
- Interface eth0 configured statically. The general syntax of a declaration is as follows:

interface name type mode

With a static configuration, you must specify various parameters using the following keywords:

- address: the IP address.
- netmask: the subnet mask.
- broadcast: the broadcast address.
- gateway: the default gateway. The /etc/hostname file contains the machine's name:

 # cat / etc / hostname
debian

V.2.2.4 Routing Configuration

With the use of the previous files and commands, there is no need to create a specific route because the default gateway is already present and routes for network interfaces are automatically set up. However, you can use the route command. Display current routes:

$ sudo route

Adds the route to network 192.168.1.0 via eth0:

$ sudo route add -net 192.168.1.0 netmask 255.255.255.0 eth0

Adds the default gateway to the router:

$ sudo route add default gw 192.168.1.1

Removes the route to network 192.168.1.0:

$ sudo route del -net 192.168.1.0

Refer to the manual (route(8)) for better understanding of this command. The route del command does not seem to work

V.2.3 Network Tools

V.2.3.1 ftp

It is useful to know the ftp command (file transfer protocol). It allows file transfer between two machines. It takes the name of the remote machine as a parameter. For the ftp command to work, the ftp service must be running on the remote machine and on port 21. Here is an example of connection.

$ ftp
ftp > open
(to) ftp.debian.org
Connected to ftp.debian.org.
220 ftp.debian.org FTP server
Name (ftp.debian.org: student) anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp > pwd
257 "/"
ftp >
Alternatively, you could connect as follows:
$ ftp ftp.debian.org
Connected to ftp.debian.org.
220 ftp.debian.org FTP server
Name (ftp.debian.org: student) anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.

Using binary mode to transfer files.
ftp > pwd
257 "/"
ftp >

The main FTP commands are described in the table below:

Table V.1 Main FTP commands

Command	Description
Open	Connect to a remote FTP server
close	Close the connection to the remote FTP server
User	Specify the username for authentication
pass	Specify the password for authentication
Pwd	Print the current working directory on the server
cd	Change the current directory on the server
Ls	List the contents of the current directory on the server
get	Download a file from the server to the local machine
Put	Upload a file from the local machine to the server
Delete	Delete a file on the server
Mkdir	Create a directory on the server
rmdir	Remove a directory from the server
quit	Close the FTP session and exit the FTP program

Telnet is a lightweight client used to open a connection and session on a remote machine that offers a Telnet server. This server is often launched from xinetd.

V.2.3.2 Installation and Service Startup

 # apt-get install telnetd xinetd

Edit the configuration file **/etc/xinetd.conf** and add the following lines:

```
service telnet
{
   flags = REUSE
   socket_type = stream
   wait = no
   user = root
   server = /usr/sbin/in.telnetd
   log_on_failure += USERID
```

```
    disable = no
    port = 23
}
```

Finally, restart the xinetd service:

 # /etc/init.d/xinetd restart

V.2.3.3 Usage

```
$ telnet 192.168.1.2
Trying 192.168.1.2...
Connected to 192.168.1.2.
Escape character is '^]'.
Debian GNU/Linux jessie/sid
debian login: etudiant
Password:
Last login: Thu Jan 9 07:59:55 CET 2014 on tty2
Linux debian 3.2.0-4-686-pae #1 SMP Debian 3.2.53-2 i686
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have new mail.
etudiant@debian:~$
```

This connects us to the remote machine where we can execute commands.

Note that the Telnet service (and client) is absolutely not secure: connections travel in clear text over the network, and any IP sniffer (such as Wireshark) can intercept and see everything that is done. Even the password is transmitted in clear text. Avoid using this service and consider using OpenSSH instead.

V.2.3.4 Ping Command

The *ping* command is a central, even essential command. The first thing one generally does to determine if a machine is accessible or not is to try to "ping" it (assuming the firewall configuration allows ICMP requests). *Ping* sends a network echo and waits for a response, the echo reply. It uses the ICMP protocol. To interrupt the command, press *[Ctrl] C*.

```
$ ping www.google.com
PING www.google.com (173.194.116.114) 56(84) bytes of data.
64 bytes from 173.194.116.114: icmp_req=1 ttl=53 time=64.8 ms
```

64 bytes from 173.194.116.114: icmp_req=2 ttl=53 time=63.7 ms
64 bytes from 173.194.116.114: icmp_req=3 ttl=53 time=63.2 ms
64 bytes from 173.194.116.114: icmp_req=4 ttl=53 time=63.1 ms
64 bytes from 173.194.116.114: icmp_req=5 ttl=53 time=63.3 ms
^C
--- www.google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 20561 ms
rtt min/avg/max/mdev = 63.162/63.694/64.897/0.636 ms

Three parameters should draw your attention:

- **-c** specifies the number of echos to send.
- **-b** allows sending an echo to a broadcast address.
- **-I** specifies the network interface.

Example

1. The **-c** parameter can be useful for testing in a script if a server responds:

 # ping -c 1 10.9.238.170 >/dev/null 2>&1 && echo "Le serveur répond"

Le serveur répond

The **-b** parameter can be used to verify that all addresses in the subnet concerned by the broadcast address respond.

 # ping -b 192.168.1.255
WARNING: pinging broadcast address
PING 192.168.1.255 (192.168.1.255) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=0.232 ms
64 bytes from 192.168.1.60: icmp_seq=1 ttl=64 time=0.240 ms
64 bytes from 192.168.1.130: icmp_seq=1 ttl=255 time=0.285 ms
64 bytes from 192.168.1.139: icmp_seq=1 ttl=255 time=0.292 ms
...

With the **-I** parameter, one can specify an outgoing card. This option is very useful for checking DNS resolution or a route.

 # ping -I eth0 192.168.1.60
PING 192.168.1.60 (192.168.1.60) from 192.168.1.10: eth0: 56(84) bytes of data.
64 bytes from 192.168.1.60: icmp_seq=1 ttl=62 time=0.478 ms
64 bytes from 192.168.1.60: icmp_seq=2 ttl=62 time=0.408 ms
....

V.2.3.5 Traceroute

When attempting to access a remote host from your machine, IP packets often travel through numerous routes, sometimes different depending on the starting and ending points,

congestion, etc. The journey goes through many gateways, which depend on their default or predefined routes. The traceroute command allows you to visualize each of the passage points of your IP packets to a given host.

$ traceroute www.debian.org
traceroute to www.debian.org (130.89.148.14), 30 hops max, 60 byte packets
1 192.168.1.1 (192.168.1.1) 4.422 ms 4.542 ms 4.658 ms
2 41.103.128.1 (41.103.128.1) 33.051 ms 33.563 ms 43.626 ms
3 172.18.0.29 (172.18.0.29) 32.074 ms 33.853 ms 34.992 ms
4 172.17.0.53 (172.17.0.53) 38.423 ms 40.142 ms 41.160 ms
5 141.170.211.9 (141.170.211.9) 93.112 ms 141.170.211.25 (141.170.211.25) 117.860 ms 141.170.211.9 (141.170.211.9) 96.047 ms
6 141.170.208.129 (141.170.208.129) 99.496 ms 141.170.208.137 (141.170.208.137) 47.718 ms 141.170.208.133 (141.170.208.133) 102.462 ms
7 ae2.jnr02.Asd001A.surf.net (195.69.144.50) 73.461 ms 74.925 ms 75.973 ms
8 AE0.500.JNR01.Asd002A.surf.net (145.145.80.81) 77.072 ms 78.547 ms 80.053 ms
9 UTwente-router.Customer.surf.net (145.145.4.46) 86.401 ms 87.931 ms 89.844 ms
10 klecker4.snt.utwente.nl (130.89.148.14) 90.803 ms 91.792 ms 104.852 ms

V.2.3.6 Whois

The whois command provides all desired information about a given domain.

$ whois debian.org
...
Domain ID: D4227449-LROR
Domain Name: DEBIAN.ORG
Created On: 10-Mar-1999 05:00:00 UTC
Last Updated On: 11-Jan-2014 09:53:04 UTC
Expiration Date: 10-Mar-2018 05:00:00 UTC
Sponsoring Registrar: Gandi SAS (R42-LROR)
Status: CLIENT TRANSFER PROHIBITED
Registrant ID: SH4438-GANDI
Registrant Name: SPI Hostmaster
Registrant Organization: Software in the Public Interest, Inc. - Debian Project
...
Registrant City: Indianapolis
Registrant State/Province: IN
Registrant Postal Code: 46250-6248
Registrant Country: US
Registrant Phone: +49.6619012303
...

V.2.3.7 Netstat

The netstat command provides a wealth of information about the network and protocols. The parameter -i provides the status of network cards to determine any potential failure or cable issues.

$ netstat -i
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0 1500 0 0 0 0 0 27 0 0 0 BMRU
eth1 1500 0 468 0 0 0 235 0 0 0 BMRU
lo 16436 0 1373 0 0 0 1373 0 0 0 LRU

By adding the parameter -e, you obtain the same result as with ifconfig -a.

$ netstat -ei
Kernel Interface table
eth0 Link encap: Ethernet HWaddr 00:27:19:b6:d6:01
inet6 addr: fe80::227:19ff:feb6:d601/64 Scope: Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:27 errors:0 dropped:0 overruns:0 carrier:27
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:4801 (4.6 KiB)
Interrupt:18 Base address:0xcc80

eth1 Link encap: Ethernet HWaddr bc:30:5b:cc:b1:dd
inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::be30:5bff:fecc:b1dd/64 Scope: Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:474 errors:0 dropped:0 overruns:0 frame:0
TX packets:238 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:93830 (91.6 KiB) TX bytes:21811 (21.2 KiB)
Interrupt:16
lo Link encap: Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope: Host
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:1398 errors:0 dropped:0 overruns:0 frame:0
TX packets:1398 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:534788 (522.2 KiB) TX bytes:534788 (522.2 KiB)

The -r parameter retrieves routing tables similar to the route command. Add the -n parameter to display IPs instead of names.

$ netstat -rn
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window irtt Iface
0.0.0.0 192.168.1.1 0.0.0.0 UG 0 0 0 eth0
192.168.1.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0

The -a parameter visualizes all connections for all protocols, including listening ports on the machine. The output is too long to fit on these pages.
 # netstat -a | wc -l
479

The -A parameter specifies the visible protocol: inet, unix, ipx, ax25, netrom, and ddp.
 # netstat -a -A inet Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
*tcp 0 0 *:sunrpc *:* LISTEN*
*tcp 0 0 *:ssh *:* LISTEN*
*tcp 0 0 *:53398 *:* LISTEN*
*tcp 0 0 localhost:ipp *:* LISTEN*
*tcp 0 0 localhost:smtp *:* LISTEN*
*udp 0 0 *:ipp *:**
*udp 0 0 *:858 *:**
*udp 0 0 localhost:893 *:**
*udp 0 0 *:41908 *:**
*udp 0 0 *:mdns *:**
*udp 0 0 *:1900 *:**
*udp 0 0 *:55994 *:**
*udp 0 0 *:sunrpc *:**

Finally, the -p parameter indicates, when possible, the PID and name of the process.

 # netstat --inet -p
Active Internet connections (without servers)
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
tcp 0 0 192.168.1.2:33908 74.125.133.106:https ESTABLISHED 3537/iceweasel

V.2.4 Genaral files

V.2.4.1 /etc/resolv.conf

The file /etc/resolv.conf is used to specify to the system which name servers and domains to query for resolving client DNS requests. The APIs are included in the library and Linux standard APIs (no need to add additional tools). This library is called the resolver.

Note that when configuring DHCP, this file is typically automatically updated and should not be modified unless DNS configuration is prohibited on your client.

$ cat /etc/resolv.conf
domain pc.dell
nameserver 192.168.1.1
nameserver 41.103.128.1

- domain: local domain name. Requests are usually truncated to relative shortcuts to the local domain. If absent, the domain name must be determined from the full hostname: it is the part after the first '.'.
- search: list of search domains. By default, when using shortcuts (short hostnames), the resolver searches the domain defined by the domain line, but here you can specify a list of domains separated by spaces or commas.
- nameserver: IP address of the name server (DNS server). You can place up to three. The resolver tries to use the first one. In case of failure (timeout), it moves to the second, and so on.
- options: options can be specified. For example, timeout:n where n indicates the timeout (in seconds) for waiting for a response from a name server before moving to the next.

V.2.4.2 /etc/hosts and /etc/networks

Even without using a name server, you can establish a mapping between IP addresses and names.

$ cat /etc/hosts
127.0.0.1 localhost
192.168.1.2 debian.pc.dell debian
...
You can do the same to name networks (which can be useful for tcp_wrappers or the route command) in the /etc/networks file.

$ cat /etc/networks
default 0.0.0.0
loopback 127.0.0.0
link-local 169.254.0.0
localnet 192.168.1.0

V.2.4.3 /etc/nsswitch.conf

The /etc/nsswitch.conf file determines the order in which the resolver (or other services) retrieves its information. The two bold lines in the example indicate that for a name

resolution request (or network), files have priority. The /etc/hosts file is read first, and then, if the resolver doesn't find the information, it goes through DNS resolution.

$ cat /etc/nsswitch.conf
passwd: compat
group: compat
shadow: compat
hosts: files mdns4_minimal [NOTFOUND=return] dns mdns4
networks: files
protocols: db files
services: db files
ethers: db files
rpc: db files
netgroup: nis

V.2.4.4 /etc/services

The /etc/services file contains the list of known network services for Linux along with the associated ports and protocols. It is used by many services (including xinetd) and subsystems like Linux's firewall. This file is indicative: it is a description and definition file; not all services listed in this file necessarily run on your machine (and thankfully given their number)! A service might be configured to listen on a different port. However, it is advisable, when adding a service that is not present in this file, to add it at the end.

$ cat /etc/services
...
ftp 21/tcp
fsp 21/udp fspd
ssh 22/tcp # SSH Remote Login Protocol
ssh 22/udp
telnet 23/tcp
smtp 25/tcp mail
...

V.2.4.5 /etc/protocols

The /etc/protocols file contains the list of protocols known by Linux.

$ cat /etc/protocols
...
ip 0 IP # internet protocol, pseudo protocol number
 # hopopt 0 HOPOPT # IPv6 Hop-by-Hop Option [RFC1883]
icmp 1 ICMP # internet control message protocol
igmp 2 IGMP # Internet Group Management
ggp 3 GGP # gateway-gateway protocol

ipencap 4 IP-ENCAP # IP encapsulated in IP (officially ``IP'')
st 5 ST # ST datagram mode
tcp 6 TCP # transmission control protocol
egp 8 EGP # exterior gateway protocol
igp 9 IGP # any private interior gateway (Cisco)
pup 12 PUP # PARC universal packet protocol
udp 17 UDP # user datagram protocol
...

V.3 Network Services xinetd

V.3.1 Presentation

The xinetd daemon is a "super-service" that controls access to a set of services. Many network services can be configured to work with xinetd, such as ftp, ssh, samba, rcp, http, etc. Specific configuration options can be applied for each managed service. When a client host connects to a network service controlled by xinetd, xinetd receives the request and first checks TCP access permissions and then the rules defined for that service (specific permissions, allocated resources, etc.). An instance of the service is then started and handed the connection. From that moment on, xinetd no longer interferes in the connection between the client and the server.

Note that the xinetd service is not installed by default on the Wheezy version of Debian. Therefore, you need to install the xinetd package before using the service.

V.3.2 Configuration

The configuration files are:

- /etc/xinetd.conf: Global configuration.
- /etc/xinetd.d/*: Directory containing service-specific files. There is one file per service, with the same name as specified in /etc/services.

$ cat /etc/xinetd.conf
 # Simple configuration file for xinetd
#
 # Some defaults, and include /etc/xinetd.d/
defaults
{
 # Please note that you need a log_type line to be able to use
 # log_on_success and log_on_failure. The default is the following:
 # log_type = SYSLOG daemon info
}
includedir /etc/xinetd.d

Content of xinetd.conf:

defaults
{
instances = 60
log_type = SYSLOG authpriv
log_on_success = HOST PID
log_on_failure = HOST
cps = 25 30
}
includedir /etc/xinetd.d

Where:

- instances: maximum number of requests a xinetd service can handle at a given time.
- log_type: in this case, logs are managed by the syslog daemon via authpriv, and logs are placed in /var/log/secure. FILE /var/log/xinetd would place logs in /var/log/xinetd.
- log_on_success: xinetd will log the event if the connection to the service succeeds. The logged information includes the host (HOST) and the PID of the server process handling the connection.
- log_on_failure: same as above but for failures. It becomes easy to know which hosts attempted to connect if, for example, the connection is not allowed.
- cps: xinetd allows only 25 connections per second to a service. If the limit is reached, xinetd will wait for 30 seconds before allowing connections again.
- includedir: includes options from files in the specified directory.

Example: /etc/xinetd.d/telnet

$ cat /etc/xinetd.d/telnet
 # default: on
 # description: The telnet server serves telnet sessions; it uses
 # unencrypted username/password pairs for authentication.
service telnet
{
disable = no
flags = REUSE
socket_type = stream
wait = no
user = root
server = /usr/sbin/in.telnetd
log_on_failure += USERID
}

Where:

- service: name of the service corresponding to a service defined in /etc/services.

- flags: attributes for the connection. REUSE indicates that the socket will be reused for a telnet connection.
- socket_type: specifies the socket type. Typically stream (tcp) or dgram (udp). Direct IP connection is made through raw.
- wait: indicates if the server is single-threaded (yes) or multi-threaded (no).
- user: under which user account the service will be launched.
- server: path to the executable to be launched.
- log_on_failure: the += indicates that we are adding the option associated with the log file in addition to the default ones. Here: login.
- disable: indicates if the service is active or not.

Some options can improve access conditions and security:

- only_from: allows access only from specified hosts.
- no_access: prevents access from specified hosts (e.g., 172.16.17.0/24).
- access_times: allows access only during a given time range (e.g., 09:00-18:30).

V.3.3 Starting and stopping services

The xinetd service is like any other service, which can be started or stopped using the service command or directly by executing /etc/init.d/xinetd.

$ sudo service xinetd start

V.4 PPP Connection

V.4.1 Modem Selection and Configuration

V.4.1.1 Winmodem Case

All analog modems connecting to a serial port (external) or emulating a real serial port (PCI card or via USB port) are fully supported under Linux. However, there is a particular category of modems called winmodems. They sometimes appear as "real" modems (they may look like them). Generally, it is advisable to avoid this type of modem. However, some models are known to work under Linux. The website http://linmodems.org/ provides information on this matter. Figure 1 gives an idea of these types of modems by showing images of some of them. Other adapters besides RTC modems are recognized by Linux as modems. This includes some ADSL adapters, as well as mobile phones connected via USB cable or Bluetooth

V.4.1.2 Peripheral Files

RS232 serial ports are named ttySn:

- /dev/ttyS0: first serial port.
- /dev/ttyS1: second serial port.
- ...

*# ls -l /dev/ttyS**
crw-rw---- 1 root dialout 4, 64 Jan 25 18:13 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 Jan 25 18:13 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 Jan 25 18:13 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 Jan 25 18:13 /dev/ttyS3

USB serial ports are named ttyUSBn: /dev/ttyUSB0, and so on. After connecting the USB-serial adapter shown in Figure 2, the dmesg command displays:

 # dmesg
[16098.567166] usb 5-1: ch341-uart converter now attached to ttyUSB0
*# ls -l /dev/ttyUSB**
crw-rw---- 1 root dialout 188, 0 Jan 25 22:41 /dev/ttyUSB0

Similarly, the kernel displays the following messages after inserting an Ooredoo 3G modem:

 # dmesg

Jan 26 22:42:12 debian64 kernel: [1486.792033] usb 3-8: new high-speed USB device number
Jan 26 22:42:12 debian64 kernel: [1486.926888] usb 3-8: New USB device found, idVendor
Jan 26 22:42:12 debian64 kernel: [1486.926896] usb 3-8: New USB device strings: Mfr=2,
Jan 26 22:42:12 debian64 kernel: [1486.926901] usb 3-8: Product: Qualcomm CDMA Technologies
Jan 26 22:42:12 debian64 kernel: [1486.926905] usb 3-8: Manufacturer: Qualcomm, Incorporated
Jan 26 22:42:12 debian64 kernel: [1486.926909] usb 3-8: SerialNumber: 000000000002
Jan 26 22:42:12 debian64 kernel: [1486.930103] option 3-8:1.0: GSM modem (1-port) converter
Jan 26 22:42:12 debian64 kernel: [1486.930263] usb 3-8: GSM modem (1-port) converter

Serial communication ports via Bluetooth are named rfcommn (for radio frequency communication): /dev/rfcomm0, and so on. To use serial ports and establish communication, you must be able to write to the devices (to send commands) and thus either have the corresponding rights, use a SUID program, or have appropriate udev rules.

FIG V.1 Examples of winmodems

FIG V.2 USB-serial adapter

V.4.1.3 Configuring the Serial Port

Serial ports are managed using the **setserial** command provided by the package of the same name. It allows querying the configuration of a serial port with the **-g** parameter. The serial port ttyS0 is of type 16550A (the fastest), uses IRQ 4, and port address 0x03f8.

 # setserial -g /dev/ttyS0 /
dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4

Since the information provided might not be sufficient, we add the **-a** parameter. This gives us, among other details, the line speed, which is 115200 bits per second.

 # setserial -a -g /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
Baud_base: 115200, close_delay: 50, divisor: 0
closing_wait: 3000

Flags: spd_normal skip_test

setserial also allows configuring the port. By querying the port with the **-G** parameter, we can find out which parameters can be adjusted and thus passed to **setserial** as command line arguments.

 # setserial -G /dev/ttyS0 /
dev/ttyS0 uart 16550A port 0x03f8 irq 4 baud_base 115200 spd_normal skip_test

From there, it is possible to configure the port for new values. For example, you can set the serial port to a speed of 57600 bps:

 # setserial /dev/ttyS0 baud_base 57600
 # setserial -ag /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
Baud_base: 57600, close_delay: 50, divisor: 0
closing_wait: 3000
Flags: spd_normal skip_test

V.4.1.4 AT Commands

All modems use a set of standard commands called AT commands. Their real name is Hayes Commands, named after the company that invented them. AT stands for Attention. The modem waits after these first letters for a sequence that allows configuration, dialing, hanging up, etc. The set is standard in principle, but the configuration varies from one model to another. If you don't want to go into detail, a generic configuration suffices and works for almost all modems. Since it's not possible to describe the AT commands here, interested readers can find them on the 3Com website (which acquired US Robotics, the best modem manufacturer): 3Com AT Commands.

Here are some examples:

- Dialing: **ATDT0102030405**
- Answering: **ATA**
- Hanging up: **ATH**

V.4.2 The PPP Protocol

The Point-to-Point Protocol (PPP) is a communication protocol used to establish network connections over serial lines, such as analog phone connections or DSL connections. It provides a reliable way to transfer data over point-to-point links and is widely used to connect users to the Internet via dial-up connections.

The PPP protocol comprises several key components:

- **Session Establishment:** PPP uses an initialization and negotiation process to establish a session between the client and server. This involves the exchange of control packets to determine connection parameters such as transmission speed, compression options, and security protocols.
- **Link Management:** Once the session is established, PPP manages the transmission of data over the link. It encapsulates data packets into PPP frames and sends them over the link. PPP can also detect and correct transmission errors to ensure data integrity.
- **Security Management:** PPP supports multiple security protocols to protect the confidentiality and integrity of data traveling over the link. This may include authentication methods such as PAP (Password Authentication Protocol) or CHAP (Challenge Handshake Authentication Protocol), as well as options for encrypting data.
- **IP Address Management:** PPP can dynamically assign IP addresses to clients using the Dynamic Host Configuration Protocol (DHCP). This allows users to automatically obtain an IP address each time they connect to the network.

In summary, the PPP protocol provides a flexible and robust solution for establishing network connections over serial lines, making it a popular choice for dial-up and DSL connections. Its support for security and IP address management makes it a versatile protocol for various network connectivity scenarios.

V.4.3 Console Connection

V.4.3.1 Manual Connection

In the following example:

- The phone number is 0102030405.
- The login is "login".
- The password is "password".
- The device is /dev/modem.

Although this could be managed by DHCP at the time of connection, it may be necessary to modify the /etc/resolv.conf file to indicate the DNS servers of the provider. The PPP connection requires that the pppd service (/usr/sbin/pppd) be executed as root. For this, the SUID right is often set:

 # ls -l /usr/sbin/pppd
-rwsr-xr-- 1 root dip 333560 Oct 19 10:59 /usr/sbin/pppd

Another solution is to give these rights to the connection tool (chat, gnome-ppp, etc.) or to modify the permissions of the devices accordingly. The configuration files are located in /etc/ppp:

```
# ls -l /etc/ppp
total 60
-rw------- 1 root root 80 Jan 14 00:51 chap-secrets
-rwxr-xr-x 1 root root 1754 Oct 19 10:59 ip-down
drwxr-xr-x 2 root root 4096 Jan 14 00:51 ip-down.d
-rwxr-xr-x 1 root root 1892 Oct 19 10:59 ip-up
drwxr-xr-x 2 root root 4096 Jan 14 00:51 ip-up.d
-rwxr-xr-x 1 root root 784 Oct 19 10:59 ipv6-down
drwxr-xr-x 2 root root 4096 Oct 19 10:59 ipv6-down.d
-rwxr-xr-x 1 root root 922 Oct 19 10:59 ipv6-up
drwxr-xr-x 2 root root 4096 Oct 19 10:59 ipv6-up.d
-rw-r--r-- 1 root root 13209 Oct 19 10:59 options
-rw------- 1 root root 1628 Jan 14 00:51 pap-secrets
drwxr-s--- 2 root dip 4096 Jan 24 16:57 peers
```

Here's an example script for establishing a PPP connection:

```
#!/bin/sh
/usr/sbin/pppd connect '/usr/sbin/chat -v ABORT ERROR ABORT "NO CARRIER" ABORT
BUSY "" ATZ OK ATDT0102030405 CONNECT "" login: "login" word: "password" '
/dev/modem 38400 noipdefault debug crtscts modem default route &
```

V.4.3.2 Using Files

Two files are required. The first will contain the commands for the pppd service, and the second will contain the communication sequence with the ISP. Both are placed in /etc/ppp/peers. Let /etc/ppp/peers/cnx1 be the first file:

```
# cat /etc/ppp/peers/cnx1
/dev/modem
connect '/usr/sbin/chat -v -f /etc/ppp/peers/cnx1-chat'
defaultroute
noipdefault
usepeerdns
115200
debug
noauth
maxfail 10
lcp -echo - interval 5
lcp -echo - failure 12
```

holdoff 3
noaccomp noccp nobsdcomp nodeflate nopcomp novj novjccomp
lock
crtscts
Each line contains at least one instruction, here are the most relevant ones:

- **/dev/modem**: the connection device (the modem),
- **connect**: the connection string sent to the ISP,
- **defaultroute**: the default route is replaced by the one provided by the ISP,
- **noipdefault**: the ISP provides the IP through its DHCP,
- **usepeerdns**: retrieves DNS information from the ISP,
- **115200**: the communication speed of the device (it will be negotiated),
- **debug**: provides detailed connection information,
- **noauth**: the ppp script does not handle authentication (see the connect line),
- **maxfail**: n connection attempts before giving up,
- **holdoff**: wait for n seconds between two connections,
- **lock**: allows exclusive access to the device file,
- **crtscts**: activates hardware flow control.

Here's the content of the second file used by the "connect" line in the first file /etc/ppp/peers/cnx1:

 # cat /etc/ppp/peers/cnx1-chat
ABORT ERROR
ABORT "NO CARRIER"
ABORT BUSY ""
ATZ
OK
ATDT0102030405
CONNECT ""
login: "login"
word: "password"

This is just an example. One must verify both from the ISP side and from the modem documentation what the correct AT commands to pass are (they are usually standard).

V.4.3.3 Connection

To initialize the connection:

 # pppd call cnx1

You should see the modem lights flashing, and if the speaker is enabled, you'll hear the characteristic noise. If the connection is established, a new network interface will appear: **ppp0**.

 # ifconfig ppp0
ppp0 Link encap:Point-to-Point Protocol
inet addr:10.xx.yy.zz P-t-P:10.xx.yy.zz Mask:255.255.255.0
UP POINTOPOINT RUNNING MTU:552 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0
TX packets:0 errors:0 dropped:0 overruns:0

V.4.3.4 Connection through a Front-End

Instead of establishing a connection from the command line and configuration files, there are several graphical tools, front-ends to PPP, that allow both configuring the modem and establishing a PPP connection with all possible network settings.

gnome-ppp is a very good example of such a tool. The main interface allows entering:

- a username,
- a password, and
- a phone number.

Clicking the "Settings" button opens a window for configuring the PPP connection. The values displayed in the various fields are automatically set by the configuration tool, after clicking the "Detect" button.

To connect to the internet, close the settings window and click the "Connect" button in the main window.

FIG V.3 The main interface of the gnome-ppp tool

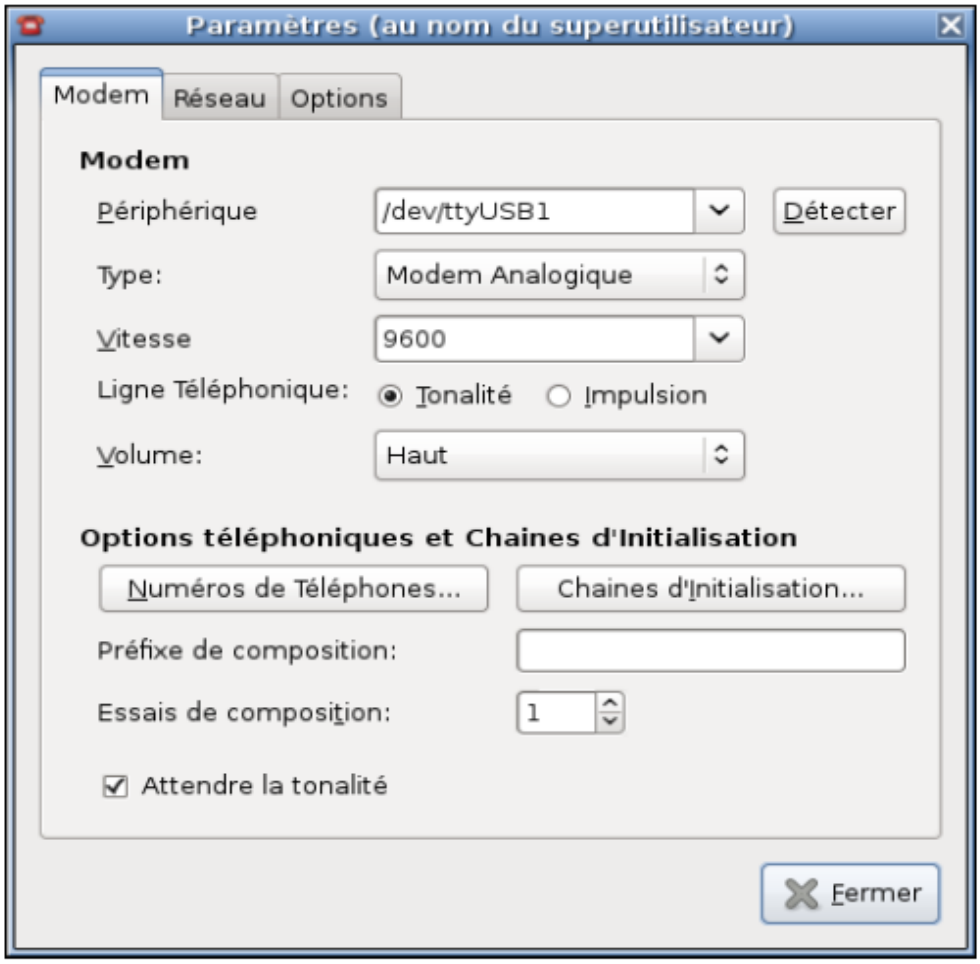


FIG V.4 Configuring a connection with gnome-ppp

FIG V.5 Connection attempt

Finally, once connected, a status window provides detailed information about the connection, its duration, and gives you the option to disconnect.

Note that the Debian distribution now comes with the NetworkManager tool, which allows for Ethernet, Wi-Fi, or PPP connections on the fly with just a few clicks.

V.5 OpenSSH

V.5.1 Overview

OpenSSH is a secure shell protocol, a mechanism that enables secure authentication, connection, and remote execution. It also allows for secure transport of the X Window protocol. In fact, it can encapsulate insecure protocols by redirecting ports. The packages to use for a server are openssh-server and openssh-client. For X, additional packages like ssh-askpass are added. The most common use is secure remote access to a machine via the ssh client.

V.5.2 Configuration

The configuration file is /etc/ssh/sshd_config. Some options may need to be modified:

- Port: the port number, default is 22.
- Protocol: set to 2,1, it allows SSH1 and SSH2. Preferably SSH2, so leave the value as 2 alone.
- ListenAddress: by default, ssh listens on all IP addresses of the server. You can allow listening only on a given interface.
- PermitRootLogin: ssh allows root connections. You can set the value to "no". In this case, you'll need to connect as a regular user and then use su.
- Banner: path to a file whose content will be displayed to users during connection. ssh is a System V service to start with service or directly via /etc/init.d/ssh.

 # service ssh start

V.5.3 Usage

The ssh command allows establishing a connection.

$ ssh -l login host
$ ssh login@host

The -X option enables X Window protocol forwarding.

$ ssh -X login@host

V.5.4 Keys and Automatic Connection

It's possible to establish automatic connection to another machine without entering a password. To do this, it's necessary, from the client user account (the machine that will connect), to generate a pair of keys, private and public. No passphrase should be entered. On the ssh server side, the client's public key must be placed in a file containing the authorized keys to connect to the destination account.

V.5.4.1 Client Side

Generate an RSA format key with the ssh-keygen command*:*

$ ssh - keygen -t rsa
Generating public / private rsa key pair .
Enter file in which to save the key (/ home / etudiant /. ssh / id_rsa):
Created directory '/ home / etudiant /. ssh '.
Enter passphrase (empty for no passphrase):
Enter same passphrase again :
Your identification has been saved in / home / etudiant /. ssh / id_rsa .
Your public key has been saved in / home / etudiant /. ssh / id_rsa . pub .
The key fingerprint is :
f6 :39:23:4 e:fa :53: d0 :4e :65:7 f:3f:fd:a3:f4 :8e:2a etudiant@p64p17bicb3

The user's directory now contains a .ssh directory

$ cd . ssh
$ ls
id_rsa id_rsa . pub
Le _chier id_rsa.pub contient la clé publique :
$ cat id_rsa . pub
ssh - rsa AAAAB3NzaC1yc2EAAAABIwAAAQEArB / VskR9v708J2EDG1LM1Q6HmKJc
P2UenurnSr7rWTSZK5w9Hzn4DCz5iMzLAPc4659I0uKJbmF3vBXozIgLrCdCZCQE
hhPLwJVLXbGNc8lMf742E / WqkkJ / uQYb31iPAU7Efosei + DVZ21No725XjiSCZ2q
zKKx7ZuNQEtxW0eVkwvlA0u7Hvrwn + FQksW3NXwTxwHhudSw7S6kIC3tyF5rkzfk
vu7zQbOGDGGPiF3aOvd0oSBNgiJtZ + M0PaoXXI3brMd66WkGfSwf4ofYKNDCA /3T
Q4xU6WxkxqTBcsjEm1gIymFAyxDo + zzf63jxLGO8Pp50DKf7DUqBx7 + rjw ==
etudiant@debian

V.5.4.2 Server Side

Navigate to the .ssh directory of the account you wish to access on the server (create it if it doesn't exist):

$ cd /home/etudiant2/.ssh

Edit the authorized_keys2 file (create it if it doesn't exist) and paste the content of the client's id_rsa.pub file onto a new line, then save it:

$ echo "ssh - rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEArB / VskR9v708J2EDG1LM1Q6HmKJcP2Uenurn
Sr7rWTSZK5w9Hzn4DCz5iMzLAPc4659I0uKJbmF3vBXozIgLrCdCZCQEhhPLwJVL
XbGNc8lMf742E / WqkkJ / uQYb31iPAU7Efosei + DVZ21No725XjiSCZ2qzKKx7ZuN
QEtxW0eVkwvlA0u7Hvrwn + FQksW3NXwTxwHhudSw7S6kIC3tyF5rkzfkvu7zQbOG
DGGPiF3aOvd0oSBNgiJtZ + M0PaoXXI3brMd66WkGfSwf4ofYKNDCA /3 TQ4xU6Wxk
*xqTBcsjEm1gIymFAyxDo + zzf63jxLGO8Pp50DKf7DUqBx7 + rjw == *

etudiant@debian " >> autorized_keys2

Attempting a connection, the password will not be requested.

$ ssh etudiant2@debian2

V.6 Setting Up a DHCP Server

The DHCP (Dynamic Host Configuration Protocol) service, a dynamic host configuration protocol, allows hosts on a network to request and receive configuration information (such as IP address, routing, DNS, etc.). There is generally only one DHCP server per network segment, although multiple servers are possible. If the server is on another segment, a DHCP relay agent can be used. In other words, a DHCP client autonomously searches for a DHCP server that will assign it an IP address. The IP address is assigned either dynamically from predefined address pools or statically based on the requester's MAC address. The information is valid for a given period of time (a lease) that can be renewed and configured. DHCP is a superset of BOOTP (Bootstrap Protocol). When the client attempts to contact a server, BOOTP provides the addressing information. DHCP manages renewals. BOOTP is based on the UDP transport protocol. A host has no network information available at startup. It must autonomously find a DHCP server. For this purpose, BOOTP broadcasts to IP 255.255.255.255 with a frame containing its information (such as its MAC address) and desired information (type of request, here DHCPDISCOVER, connection port, etc.). The broadcast is sent by definition to all hosts on the local network. When the DHCP server detects the frame, it also broadcasts (the client host does not yet have an IP address) with the basic information desired by the host (DHCPOFFER, initial parameters). The host establishes an initial configuration and then requests confirmation of the IP (DHCPREQUEST). The DHCP server confirms (DHCPACK). The lease is confirmed, and the client then has all valid information.

V.6.1 Starting DHCP Server

The isc-dhcp-server (provided by the package of the same name) is a service (daemon) launched using a script (/etc/init.d/isc-dhcp-server). It is configured using the file /etc/dhcp/dhcpd.conf. The allocated IP addresses are stored in */var/lib/dhcp/dhcpd.leases.*

service isc -dhcp - server start
or
 # / etc / init .d/isc -dhcp - server start

V.6.1.1 Basic Information

The configuration file for a server, if you are sticking to basic settings, is quite simple.

```
ddns-update-style none; # no DNS update by DHCP
option domain-name "toto.fr"; # domain name transmitted to the client
option domain-name-servers 192.168.1.254; # list of DNS servers separated by commas
default-lease-time 21600; # default lease duration in seconds without explicit request (21600 s = 6h)
max-lease-time 43200; # maximum lease duration if client request is higher (43200 s = 12 h)
```

As isc-dhcp-server can handle multiple subnets, you must specify the rules to apply for each subnet. Generally, in the context of a small network, only one block will be present, but all cases are possible. If you are certain that there is only one network, you can omit the subnet declaration.

```
# Managing the subnet 192.168.1.0
subnet 192.168.1.0 netmask 255.255.255.0
{
   option routers 192.168.1.254; # gateway for this network
   option subnet-mask 255.255.255.0; # subnet mask
   range 192.168.1.2 192.168.1.250; # DHCP range configuration
   # Static IP assignments
   host station1
   {
      hardware ethernet 00:A0:ad:41:5c:b1; # MAC address
      fixed-address 192.168.1.1; # this machine will have IP 192.168.1.1
   }
}
```

Some DHCP clients completely ignore the fact that a DHCP server can dynamically allocate a hostname to the host. In the previous example, the machine with IP 192.168.1.1 should obtain the name station1. Here's an example:

```
# hosts will be assigned names of declared hosts
use-host-decl-names on;
host station1
{
   hardware ethernet 00:A0:ad:41:5c:b1; # MAC address
   fixed-address 192.168.1.1; # this machine will have IP 192.168.1.1
   # and the name station1
}
```

You can also work on a case-by-case basis:

```
host station2
{
   hardware ethernet 00:A0:ad:41:5c:b2; # MAC address
   fixed-address 192.168.1.251; # this host will have IP 192.168.1.251
   option host-name "station2"; # this host will have the name station2
}
```

V.6.1.2 Client Side

The dhclient client (provided by the isc-dhcp-client package) enables DHCP activation on a network interface. The simplest method is, for example, for eth0:

```
# dhclient eth0 &
```

Options can be passed to dhclient to handle various possibilities. Among these options:

- -D: allows modification of the domain name,
- -H: allows modification of the hostname,
- -R: avoids overwriting the resolv.conf file,
- -l: allows modification of the leasetime (in seconds).

```
# dhclient -D -H -l 86400 eth0
```

Bibliography

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Operating System Concepts, 10th Edition, Wiley, 2021.

- Greg Gagne, Peter B. Galvin, Peter Galvin, Avi Silberschatz, Applied Operating System Concepts, Wiley, 2003.

- Andrew Tanenbaum, Herbert Bos ,Modern Operating Systems, 4 th edition, Prentice Hall, New Jersey, 2014.

- Ethem Mining, Linux for Beginners: A Practical and Comprehensive Guide to Learn Linux Operating System and Master Linux Command Line. Contains Self-Evaluation Tests to Verify Your Learning Level Paperback, Everooks Ltd , 2020.

- Ida M. Flynn, Ann McIver-McHoes, Understanding Operating Systems, Third Edition, Cengage Learning, 2000.

- Mary S. Gorman; S. Todd Stubbs, Introduction To Operating Systems, A Survey Course, Course Technology, 2000.

- William S. Davis, Operating Systems: A Systematic View, 4th edition, Addison-Wesley, 1992.

- Andrew Tanenbaum, Albert Woodhull, Operating Systems Design and Implementation, Pearson, 2006.