University of Ljubljana
Faculty of Electrical Engineering

JANEZ PUHAN

Operating Systems, Embedded Systems, and Real-Time Systems

FE Publishing

University of Ljubljana
Faculty of electrical engineering

Operating Systems, Embedded Systems, and Real-Time Systems

Janez Puhan

Ljubljana, 2015

CIP - Cataloging In Publication
National and University Library, Ljubljana

004.451(078.5)(0.034.2)

PUHAN, Janez, 1969-
Operating Systems, Embedded Systems, and Real-Time Systems [Electronic source] / Janez Puhan = [editor] Faculty of Electrical Engineering. - 1st ed. - El. book. - Ljubljana : FE Publishing, 2015

Access method (URL): http://fides.fe.uni-lj.si/~janezp/operating_systems_embedded_systems_and_real-time_systems.pdf

ISBN 978-961-243-275-1 (pdf)

278131456

URL: http://fides.fe.uni-lj.si/~janezp/operating_systems_embedded_systems_and_real-time_systems.pdf

Janez Puhan
University of Ljubljana, Faculty of Electrical Engineering
SI-1000 Ljubljana, Tržaška cesta 25
janez.puhan@fe.uni-lj.si

Publisher: FE Publishing, Ljubljana
UL Faculty of Electrical Engineering, Ljubljana
Editor: prof. dr. Sašo Tomažič

Reviewers: prof. dr. Tadej Tuma, prof. dr. Patricio Bulić
1st edition

Contents

Preface

The following text represents a real-time operating-system course textbook. The course is held in the third semester of the Master's study program in Electrical Engineering at the Faculty of Electrical Engineering of the University of Ljubljana, Slovenia. It introduces the students of Electronics into the operating systems and real-time concepts having the embedded systems perspective in mind.

Although the covered mechanisms and principles are general, they are given through Linux operating system and POSIX application programming interface examples. An important part of the course is the hands-on laboratory work where the examples can be carried out. The Phytec's phyCORE-i.MX27 development kit with the Freescale's i.MX27 microcontroller is used as an embedded system platform.

The textbook is a kind of a crash course. The topics are explained by examples providing a flying start to a beginner. The reader should consult other sources for a detailed explanation.

The first three chapters describe the operating system and network configuration basic principles. In Chapter 1, the students get familiar with the operating system parts, common Linux commands and program compiling. Chapter 2 describes the fundamentals of the network structure. In Chapter 3, a brief description of the graphical user interface with the X window system is presented.

The focus of Chapter 4 is on the Phytec's phyCORE-i.MX27 embedded system platform. It describes installation of the embedded Linux operating system, booting with the Barebox boot-loader and working with some peripheral devices (framebuffer, touchscreen, serial port, ethernet, etc.), and gives examples of a graphical application written in the Qt, cross-compilation and remote debugging.

Chapter 5 deals with the real-time properties and how to achieve them in Linux. A real-time application code skeleton is drafted by shedding light on various aspects, such as priority, scheduling policy, stack and heap memory page faults, etc.

In Chapters 6 and 7, the inter-process communication and simultaneous access are discussed. Various communication techniques are presented by the POSIX-compliant C code examples. The same approach is continued with the resource-access techniques. The circumstances leading to deadlock situations with possible solutions are presented.

Chapter 8 enables a glimpse into kernel programming and provides a small tutorial of programming, compiling and cross-compiling the "Hello World!" kernel module.

The textbook is available in pdf format on the Internet at *http://fides.fe.uni-lj.si/~janezp/operating_systems_embedded_systems_and_real-time_systems.pdf*. Also the source code of the examples in the textbook is available at *http://fides.fe.uni-lj.si/~janezp/operating_systems_embedded_systems_and_real-time_systems__code.zip*.

Chapter 1

Operating system

An operating system is a suite of programs and data making a computer work (e.g. managing the hardware resources, providing services for application programs, etc.). Linux [1, 2] refers to the family of the Unix-like [3] computer operating systems using the Linux kernel. The Linux operating systems are made up of three parts:

- kernel,
- shell and
- programs.

1.1 Operating-system parts

1.1.1 Kernel

At the computer boot, BIOS (Basic Input/Output System) performs start-up tasks to recognize and start the hardware. Then it loads and executes the partition boot code from the designated boot device (e.g. hard disk) containing the first stage of the bootstrap loader or shortly the boot-loader. A first-stage boot-loader is a small program that loads the more complex second-stage boot-loader code into RAM (Random-Access Memory) and starts it. A second-stage boot-loader, such as GRUB (GRand Unified Bootloader) or LILO (LInux LOader), loads a kernel and transfers execution to it. The second-stage boot-loaders usually can be configured to give a user multiple booting choices. These choices can include different operating systems, different versions of the same operating system, different operating-system loading options or standalone programs that can run without an operating system (e.g. memory test programs, games, etc.). A second-stage boot-loader configuration file (e.g. `/boot/grub/grub.cfg` for GRUB and `/etc/lilo.conf` for LILO) contains information about the kernel location, options, etc.

A kernel [4] is the center of the operating system. It allocates the memory and CPU (Central Processing Unit) time to programs, handles the file storage and communications, responds to system calls, etc. Traditionally, the kernel image on the Unix platforms is stored in the `/unix` file. The kernels that support the virtual memory feature have the `vm` prefix (`/vmunix`). The linux kernel (`/vmlinux`) can usually be found in a statically linked, `/vmlinuz` executable file, where the letter z at the end denotes that it is compressed (zipped).

The kernel initializes the hardware, mounts the root file system (see subsection 1.2.3), starts the operating system scheduler and the first process called `init` (`/sbin/init`). Then it goes idle. The `init` process spawns all other processes. It sets up all the non-operating system services and structures in order to create a user environment (e.g. `ftp` (File Transfer Protocol) and `ssh` (Secure SHell) services, `getty` (GET TeletYpe) text login program, `gdm` (GNOME (GNU (GNU's Not Unix) Object Model Environment) Display Manager) GUI (Graphical User Interface) login program, etc.). The `init` process runs as a daemon and typically has PID (Process IDentifier) 1. The daemon process is a process that runs in a background.

1.1.2 Shell

When a user logs in, a login program (e.g. `/sbin/getty` (or an equivalent process in a graphical environment) started by the `init` process) checks the username and password and then starts another program called shell. A shell provides an interface between the user and the kernel [5]. The command line shell is a CLI (Command Line Interpreter), while the graphical shells provide a GUI. The shell interprets the user commands and arranges for them to be carried out. The commands themselves are programs. When they terminate, the shell gives the user another prompt (e.g. `user@host:working_directory$`).

As an illustration of the way the shell and kernel work together, suppose a user types the `rm myfile` command (which has the effect of removing the file `myfile`). The shell searches for the file containing the `rm` program and then requests the kernel, through the system calls, to execute the `rm` program on `myfile`. When the `rm myfile` process finishes running, the shell returns the prompt to the user, indicating that it is waiting for further commands.

To make typing of the commands easier, most command line shells provide the file completion and history features. By typing a part of the name of a command, filename or directory and pressing the `Tab` key, the shell will complete the rest of the name automatically. If the shell finds more than one name beginning with the given letters, it will do nothing on the first `Tab` and will display the possibilities on the second `Tab` stroke. The shell also keeps a list of the previous commands that can be displayed by the `history` command. To repeat a command, use the cursor keys to scroll up and down the history list.

The command line shells can be closed either by executing the `exit` command or by pressing `Ctrl-D`.

1.1.3 Programs

The programs are executable files providing common services. They are considered as a part of the operating system. In Linux, they reside in the `/sbin`, `/bin`, `/usr/sbin` and `/usr/bin` directories (e.g. the `rm` program executing the `rm` command can be found in `/bin/rm`).

1.2 File storage

Everything in Linux is either a file or a process. In this section, a few words about the files follow. The files are grouped together in a directory structure. It is a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called the root directory `/`. All the files and directories appear under the root directory. A part of a typical Linux directory structure is shown in Fig. 1.1.

According to Fig. 1.1, the `/boot` directory contains the `/boot/grub` subdirectory and `vmlinuz-2.6.32-5-686` file. Note that the `/vmlinuz` file is a symbolic link to the `/boot/vmlinuz-2.6.32-5-686` file. The symbolic link indicates the physical location of the file in the directory structure. As indicated, each file or

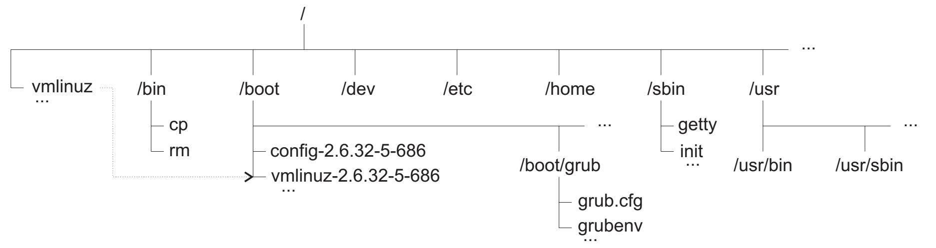

Figure 1.1: Part of a standard Linux directory structure

directory location can be described by a full path from the root directory downwards. A full path is often referred to as absolute path. Besides the absolute path, a location can also be given by a relative path describing a path to a file or directory from the current working directory (see subsection 1.3.1).

Linux is, as all the Unix operating systems are, case-sensitive. So the `data.txt`, `Data.txt` and `DATA.txt` files are three separate files. The same applies to the commands; `RM` is not the same as `rm`.

1.2.1 Directory structure

FHS (File system Hierarchy Standard) [6, 7] defines the main directories and their contents in Linux, although some distributions do not follow FHS completely. A short list of the most important directories with a description follows:

`/`	primary hierarchy root and root directory of the entire file system hierarchy
`/bin`	essential command binaries (e.g. `/bin/rm`) needing to be available in a single user mode available to all users
`/boot`	boot loader files (e.g. kernel - `/boot/vmlinuz-2.6.32-5-686`)
`/dev`	essential devices (e.g. hard disk - `/dev/sda`, terminal - `/dev/tty0`, to nothing - `/dev/null`, from nothing - `/dev/zero`)
`/etc`	host-specific system-wide static configuration files (e.g. `/etc/passwd`)
`/etc/X11`	configuration files for the X Window System version 11
`/home`	users' home directories
`/lib`	libraries essential for binaries in `/bin` and `/sbin`
`/lost+found`	parts of the restored files at the file system check (e.g. at `fsck` command)
`/media`	removable media (e.g. CD-ROM) mounting points
`/mnt`	temporarily mounted file systems
`/opt`	optional application software packages
`/proc`	process file system (`procfs`) mounting point; documenting the kernel and process status as the text files (e.g. the `/proc/cmdline` file contains kernel parameters at boot, the `/proc/1` directory contains information about the `/sbin/init` process (PID = 1))
`/root`	root user home directory
`/sbin`	essential system binaries (e.g. `/sbin/init`, `/sbin/getty`)
`/srv`	data served by the system (e.g. directory with the files served by TFTP (Trivial File Transfer Protocol) - `/srv/tftp`)
`/sys`	system file system (`sysfs`) mounting point; documenting the device status as text files (e.g. the `/sys/block/sda`

	directory contains information about the hard disk)
`/tmp`	temporary files not preserved at a reboot
`/usr`	majority of utilities and applications
`/usr/bin`	non-essential command binaries
`/usr/include`	standard C include files (e.g. `stdio.h`)
`/usr/lib`	libraries for binaries in `/usr/bin` and `/usr/sbin`
`/usr/local`	locally installed software, software not installed from the packages
`/usr/sbin`	non-essential system binaries
`/usr/share`	shared data
`/usr/src`	source code (e.g.; kernel source code with its header files)
`/var`	variable files expected to continually change during normal operation of the system
`/var/cache`	cache data locally generated as a result of time-consuming operations
`/var/lib`	data (e.g. databases)
`/var/lock`	lock files keeping track of resources shared by multiple applications
`/var/log`	various log files
`/var/mail`	users' mailboxes
`/var/run`	contains running system information since the last boot (e.g. currently logged-in users, running daemons)
`/var/spool`	data waiting for processing (e.g. print queues, unread mail)
`/var/tmp`	temporary files preserved at a reboot

1.2.2 Storage devices

As the data storage devices, the hard disk drives are usually used. A hard disk drive is divided into MBR (Master Boot Record) and partitions. MBR is the first sector of a hard disk drive containing a first-stage boot-loader code (see subsection 1.1.1) and a partition table. Each hard disk drive has the MBR although it is not used at the computer boot and does not contain the boot-loader code.

Partitions are logical storage units used to treat one physical hard disk drive as multiple disks. The first sector of each partition is the partition boot sector containing the partition boot-loader. The partition boot sector is also called VBR (Volume Boot Record). The MBR boot-loader finds the partition marked as a bootable and loads that partition boot-loader that starts the operating system code. There can be only four partitions, i.e. the primary or physical partitions. Extended partitions are introduced to allow more. An extended partition is a primary partition divided into sub-partitions. The sub- or logical partitions are used in the same way as the pure primary partitions.

In Linux, the hard disks and partitions, as other devices, are represented as files in the `/dev` directory (e.g. `/dev/sda` - first hard disk drive, `/dev/sda1` - first partition on the first hard disk drive, `/dev/sda2` - second partition on the first hard disk drive, `/dev/sdb` - second hard disk drive, etc.).

1.2.3 Partitions and file systems

To store files onto a partition, it has to be formatted. In other words, it has to contain a file system. A file system organizes the files and directories, keeps track of which areas of the media correspond to which file and which are not being used, etc. There are multiple types of the file systems (e.g. NTFS (New Technology File System) is usually used by the Windows operating systems, FAT32 (32-bit File Allocation Table) is usually used by the USB (Universal Serial Bus) keys, and ext3 (third extended file system) is usually used by default in Linux, etc.)

The journaling file system is a type of the file system that keeps track of changes that will be made in a journal before committing them to the file system [8]. A journal is usually a circular log in a dedicated area of the file system. Because of the described feature, a journaled file system is less likely to become corrupted at a system crash or power failure (e.g. ext3 is a journaled file system, while NTFS and FAT32 are not).

A virtual file system is an abstraction layer on top of a more concrete file system (e.g. `procfs` and `sysfs` are virtual file systems mounted at the `/proc` and `/sys` directories, see subsection 1.2.1).

Linux can work with many hard disk drives, many different partitions and file systems at the same time in one single directory structure (see subsection 1.2.1). A file system that contains a root directory (`/`) is called a root file system. Another file system can be mounted at an arbitrary directory or mounting point. Mounting a file system simply means making a particular file system accessible at a certain point in the directory structure (e.g. `/home` is often a separate partition mounted at the `/home` directory in a root file system. That means that the top directory in this separate partition can be accessed at the `/home` location in the directory structure. The same usually applies to `/boot` which is also often a separate partition. Sometimes, `/var` is a separate partition, too.). The root file system is mounted at the computer boot by a kernel (see subsection 1.1.1).

1.3 Login and basic commands

A common user interface to the computer consists of a display and keyboard (and mouse). It is called a console or a terminal. A display and keyboard (and mouse) physically attached to the computer are called the system console. Besides the system console, Linux provides also virtual consoles. The system console (actual display and keyboard (and mouse)) can be used to switch between the multiple virtual consoles to access the unrelated user interfaces.

Usually, Linux has six virtual text consoles or terminals with a login prompt to a shell and one virtual graphical console with a login screen to GUI. To switch to the first virtual console, press `Ctrl-Alt-F1`, to switch to the second virtual console, press `Ctrl-Alt-F2`, etc. The first six virtual consoles are textual, the seventh is graphical.

The user logs into Linux by typing her/his username and password. After the login procedure on a text terminal, a command line shell (see subsection 1.1.2) responds with a prompt indicating that the shell is prepared for the next command [9]. In the graphical terminal, GUI is started. The command line shell can be started in GUI by the Terminal program which opens a command line shell in a window. There are several shell programs available. Linux usually uses bash (Bourne Again SHell, a successor of bsh (Bourne SHell)). The shell program is terminated by the `exit` command or by pressing `Ctrl-D` representing the `EOF` (End Of File) character. A shell termination automatically means logout on the text terminal, while on the graphical terminal only the Terminal window is closed.

The shell signs its readiness to accept a new command with a command prompt. The command prompt is a sequence of characters in the command line shell usually including the information about the user, host machine and current working directory. In Linux, it typically ends with the `$` character for a normal user and `#` character for a super user or root user.

`user@host:working_directory$`	normal user prompt
`root@host:working_directory#`	super user prompt

1.3.1 Naming conventions, special directory names and wildcards

A filename is a sequence of characters used to identify a file. The filenames in Linux can contain any character other than the forward slash (`/`) and null character. Spaces are permitted. The signs, such as the dollar sign (`$`), percentage sign (`%`) and brackets (`{}`), are also permitted but not recommended because of their special meanings to the shell. The filename must be unique within a directory. However, multiple files and directories with the same name can reside in different directories. Such files have different absolute paths, thus enabling the system to distinguish them.

~ (home directory)
After the login procedure, the user finds her/him in her/his home directory which contains the user's files. In Linux, the normal user's home directory is named with the username and is placed in the `/home` directory (e.g. `/home/username`). An exception is a super user whose home directory is `/root`. The current (normal or super) user's home directory can also be marked with the ~ character (e.g. the `root@host:~#` prompt indicates that the current working directory is `/root`). Home directory of another user can be obtained by `~username`.

`.` (current working directory)
The dot character denotes the current working directory used when a file is referred to by a relative path (as opposed to a file designated by a full path from a root (`/`) directory).

`..` (parent directory)
The parent directory is a directory above the current directory (`.`). That is a directory in which the current directory (`.`) is located. In the current directory full or absolute path, the parent directory is a predecessor of the current directory. The parent directory of the root directory is the root directory.

`*` (asterisk wildcard)
The asterisk wildcard represents any sequence of characters including none (e.g. `*.txt` means all files ending with the `.txt` extension).

`?` (question mark wildcard)
The question mark wildcard represents one arbitrary character (e.g. `*.???` means all files ending with a three-character extension).

1.3.2 Files and directories

A command consists of a command name and arguments. The arguments are usually optional. If an argument begins with the `-` character, it is called a parameter or an option (e.g. `ls -l`).

`ls` (list)

The `ls` command lists the contents of a directory. It is a program in the `/bin` directory.

Examples:

`ls`	list the current working directory
`ls -l`	list the current working directory in a long format (with details)

	(with details)
`ls /home`	list the `/home` directory
`ls -d *.dat`	list the current working directory for the `.dat` files/directories, for directories print only the names not the contents
`ls -a`	list all files in the current working directory including those whose names begin with a dot
`ls ~`	list the user's home directory

`cd` (change directory)

The `cd` command changes the current working directory. It is a part of the bash.

Examples:

`cd mydir`	change to the `mydir` directory
`cd ..`	change to the parent directory
`cd`	change to the home directory
`cd ../somedir`	change to the `somedir` directory residing in the parent directory

`pwd` (print working directory)

The `pwd` command displays the name of (absolute path to) the current working directory. It is a program in the `/bin` directory.

Example:

`pwd`	print the absolute path to the working directory

`cp` (copy)

The `cp` command creates a copy of a file. It is a program in the `/bin` directory.

Examples:

`cp file1 file2`	copy `file1` into `file2`, both in the current directory
`cp data dir`	copy the `data` file in the current directory into the `dir` subdirectory
`cp /home/user/data /home/user/backup/data`	copy the `data` file in the `/home/user` directory into a file with the same name in the `/home/user/backup` subdirectory
`cp *.txt dir`	copy all the `.txt` files into the `dir` subdirectory
`cp -r /home/user/data/* /home/user/backup`	copy all files and directories in the `/home/user/data` directory into the `/home/user/backup` directory

`dd` (data duplication)

The `dd` command performs a low-level copy of a part of a file or device. It is a program in the `/bin` directory.

Examples:

`dd if=file1 of=file2`	copy `file1` (`if` - input file) into `file2` (`of` - output file), same as the `cp` command
`dd if=/dev/sda of=mbr bs=512 count=1`	create an MBR image (first 512 bytes; `bs` - block size, `count` - number of blocks) of the `/dev/sda` disk into the `mbr` file
`dd if=sect of=/dev/sda ibs=512 skip=3 count=1 obs=1k seek=5`	copy 1537$^{\text{th}}$ to 2048$^{\text{th}}$ byte of the `sect` file (`ibs` - input block size) into 5121$^{\text{th}}$ to 5632$^{\text{th}}$ byte of `/dev/sda` (`obs` - output block size)

If logged as a super user, handle the `dd` command with an extreme care especially when the output file is a device (i.e. hard disk). A misuse can cause loss of data (e.g. corruption of the partition table).

`mv` (move)

The `mv` command renames a file and/or moves it from one directory to another. It is a program in the `/bin` directory.

Examples:

`mv data dir`	move the `data` file in the current directory into the `dir` subdirectory
`mv data1 ../data2`	move the `data1` file to the parent directory and rename it into `data2`

`rm` (remove)

The `rm` command deletes a file. It is a program in the `/bin` directory.

Examples:

`rm data`	delete the `data` file without prompting for confirmation
`rm -r dir`	remove the `dir` directory with all files and subdirectories in it, prompt for removal of the write-protected files
`rm -r -f dir`	remove the `dir` directory with all files and subdirectories in it without prompting for confirmation

`mkdir` (make directory)

The `mkdir` command creates a new directory. It is a program in the `/bin` directory.

Example:

`mkdir mydir`	create a new directory called `mydir`

`rmdir` (remove directory)

The `rmdir` command deletes an empty directory. It is a program in the `/bin` directory.

Example:

`rmdir dir`	remove the `dir` directory if empty

`clear` (clear)

The `clear` command clears the text terminal screen or Terminal window. It is a program in the `/usr/bin` directory.

Example:

`clear`	clear the screen

`cat` (concatenate)

The `cat` command displays the contents of a file on the screen or concatenates the files. It is a program in the `/bin` directory.

Examples:

`cat data`	display the contents of the `data` file
`cat file1 file2 > file3`	concatenate the `file1` and `file2` files into the `file3` file (see section 1.4)

`less` (less)

The `less` command displays the contents of a file on the screen in its own environment. Listing with the cursor up and down, page up and page down keys is enabled. It is a program in the `/usr/bin` directory. To terminate the `less` environment, press `q`.

Example:

`less data`	display the contents of the `data` file
`less -N data`	display the contents of the `data` file with the line numbers

Searching for a particular phrase or sequence of characters is possible. To find all the occurrences of a phrase consisting for instance of two words, i.e. `word1 word2`, type `/word1 word2` in the `less` environment. No quotes are needed. The space character is treated as any other character. To display the next occurrence of the searched phrase, type `n`.

`head` (head)

The `head` command displays the first ten (by default) lines of a file on the screen. It is a program in the `/usr/bin` directory.

Examples:

`head data`	display the first ten lines of the `data` file
`head -5 data`	display the first five lines of the `data` file

`tail` (tail)

The `tail` command displays the last ten (by default) lines of a file on the screen. It is a program in the `/usr/bin` directory.

Examples:

`tail data`	display the last ten lines of the `data` file
`tail -5 data`	display the last five lines of the `data` file
`tail -f log`	display the last ten lines of the `log` file, update as new lines are being added to `log`, to terminate, press `Ctrl-C`
`tail -f log | grep sometext`	monitor the lines added to the `log` file, filter out the lines containing `sometext` (see subsections 1.3.3 and section 1.4), to terminate, press `Ctrl-C`

`mount` (mount)

The `mount` command attaches (i.e. mounts) a file system to a specified mounting point. It is a program in the `/bin` directory.

Examples:

`mount`	list all mounted file systems
`mount -t ext3`	list the mounted file systems of the `ext3` type
`mount /dev/sda2 /mnt/second`	mount the second partition of the `/dev/sda` hard-disk drive to the `/mnt/second` mounting point; the file system type is guessed if possible (can be performed only by a super user)
`mount -t ext3 /dev/sda2 /mnt/second`	mount the second partition of the `/dev/sda` hard-disk drive containing a file system of the `ext3` type to the `/mnt/second` mounting point (can be performed only by a super user)

`umount` (unmount)

The `umount` command detaches (i.e. unmounts) a file system mounted to a specified mounting point. It is a program in the `/bin` directory.

Example:

`umount /mnt/second`	unmount the file system mounted to `/mnt/second`

`fdisk` (fixed disk)

The `fdisk` command is a partition table manipulator. It is used for creating and modifying partitions, assigning a file system to a partition, etc. The `fdisk` command can be executed only by a super user. It is a program in the `/sbin` directory.

Example:

`fdisk /dev/sda`	start the menu-driven partition-table manipulation utility on the `/dev/sda` hard-disk drive

`mkfs` (make a file system)

The `mkfs` command builds a file system on a given partition. The `mkfs` command can be executed only by a super user. It is a program in the `/sbin` directory.

Examples:

`mkfs /dev/sda2`	create a file system of the default type (i.e. `ext2`) on the second partition of the `/dev/sda` hard-disk drive
`mkfs -v /dev/sda2`	create a file system of the default type (i.e. `ext2`) on the second partition of the `/dev/sda` hard-disk drive and print the progress information
`mkfs -t ext3 /dev/sda2`	create the `ext3` type file system on the second partition of the `/dev/sda` hard-disk drive; `mkfs` is in fact a front-end for various file system builders (e.g. `mkfs.ext3`)

1.3.3 Searching

`grep` (global regular expression print)

The `grep` command searches for a particular phrase or a sequence of characters in a file. It is a program in the `/bin` directory. The following characters have a special meaning in the search description:

`.`	any single character
`*`	arbitrary characters
`[]`	any single character listed within brackets

`^`	beginning of a line
`$`	end of a line
`\\`	escape sequence for special characters

Examples:

`grep 'pattern' *.txt`	search all the `.txt` files in the current directory and print all lines containing the `pattern` word
`grep 'pattern' /home/user/data/*`	search all files in the `/home/user/data` directory and print all lines containing a `pattern`
`grep -l 'pattern' /home/user/data/*`	print the filenames of the files in `/home/user/data` containing the `pattern`
`grep -i 'pattern' data`	word print all lines in the `data` file containing the case-insensitive `pattern` word
`grep -v 'pattern' data`	print all lines in the `data` file not containing the word `pattern`
`grep -n 'pattern' data`	print all lines in the `data` file containing the word `pattern` with the line numbers
`grep -c 'pattern' data`	print the number of lines in the `data` file containing the `pattern` word
`grep 'j..k' data`	print all lines in the `data` file containing a four-character word beginning with `j` and ending with `k` (e.g. `jack`)
`grep 'j*k' data`	print all lines in the `data` file containing a word beginning with `j` and ending with `k` (e.g. `jack`, `jailbreak`)
`grep 'j[ao]ck' data`	print all lines in the `data` file containing the word `jack` or `jock`
`grep '^jack' data`	print all lines in the `data` file starting with the word `jack`
`grep 'jack$' data`	print all lines in the `data` file ending with the word `jack`
`grep '\\[jack\\]' data`	print all lines in the `data` file containing the sequence `[jack]`

`wc` (word count)

The `wc` command counts the lines, words and characters (bytes) in a file. It is a program in the `/usr/bin` directory.

Examples:

`wc data`	count the lines, words and bytes of the `data` file
`wc -l data`	count the lines in the `data` file
`wc -w data`	count the words in the `data` file
`wc -c data`	count the bytes in the `data` file

`find` (find)

The `find` command finds the files, directories, etc., matching a given name pattern. It searches the entire directory structure beneath one or more given directories. All subdirectories are included recursively. The find command is a program in the `/usr/bin` directory.

Examples:

`find /usr . -name '*.txt' -type f`
: search `/usr` and current directory for the files ending with `.txt`

`find . -name 'data' -type d`
: search the current directory for the `data` directories

`find . -not -name '*.txt' -type f`
: search the current directory for the non `.txt` files

`find . -name '*.txt' -type f -exec grep -l 'pattern' {} \;`
: search the current directory for the `.txt` files and print the filenames of those containing the `pattern` word

`find . -name '*.dat' -exec ls -ld {} \;`
: search the current directory for the `.dat` files, directories, etc., and print their names in a long format

`find . -name '*.dat' -exec rm -r {} \;`
: search the current directory for the `.dat` files, directories, etc., and remove them

In the last three examples, the `-exec` option is used to execute another command on the found results. The list of the files, directories, etc., found by the `find` command is passed to the following command at the `{}` placeholder. The command is executed for each item in the list and is ended by `\;`.

1.3.4 File compressing

`gzip` (GNU zip)

The `gzip` command compresses a file. A compressed file has a `.gz` extension. The original file is removed. It is a program in the `/bin` directory.

Example:

`gzip data` compress the `data` file

`gunzip` (GNU unzip)

The `gunzip` command uncompresses a compressed file. The original file is restored. The compressed file is removed. It is a program in the `/bin` directory.

Example:

`gunzip data.gz` uncompress the `data.gz` file `zcat` (zip concatenate)

The `zcat` command expands a compressed file to a standard output. It is a program in the `/bin` directory.

Example:

`zcat data.gz`	print the contents of the `data` file

`tar` (tape archiver)

The `tar` command creates or expands a collection of files within a single file. A collection of files usually has a `.tar` extension. If a collection is also compressed, it is called a tarball. The `.tar.gz` or `.tgz` extension is used in that case. It is a program in the `/bin` directory.

Examples:

`tar -cvf data.tar ./dir/*.dat`	in the current directory create the `data.tar` collection containing all `.dat` files in the `dir` subdirectory
`tar -cvzf data.tar.gz ./dir`	in the current directory create the `data.tar.gz` compressed collection containing the entire `dir` subdirectory
`tar -xvf data.tar`	expand the `data.tar` collection file
`tar -xvzf data.tar.gz`	expand the `data.tar.gz` compressed collection file

1.3.5 Miscellaneous commands

`date` (date)

The `date` command retrieves or sets the system date and time. It is a program in the `/bin` directory.

Examples:

`date` print the system date and time

`date -s "12/20/2012 23:59:59"`
set a new system date and time
(can be performed only by a super user)

`date '+DATE: %m/%d/%y%nTIME:%H:%M:%S'`
print the system date and time in the specified format
(`%n` stands for a new line)

`df` (disk free)

The `df` command reports the amount of the used and free disk space for every mounted file system. It is a program in the `/bin` directory.

Examples:

`df`	print the space information for the mounted file systems
`df -h`	print the space information for the mounted file systems in a human-readable format

`du` (disk usage)

The `du` command reports the size of each subdirectory in kB. It is a program in the `/usr/bin` directory.

Examples:

`du *.dat`	print the size in kB for each `.dat` file and each subdirectory in all directories in the current working directory whose name ends with `.dat`
`du -s -h *.dat`	print the summary size of all `.dat` files and subdirectories in a human-readable form

`echo` (echo)

The `echo` command displays its argument on the screen. It is a program in the `/bin` directory.

Examples:

`echo Hello world`	print the `Hello world` to the screen
`echo $PATH`	print the value of the `PATH` environment variable (see section 1.7)

`exit` (exit)

The `exit` command terminates the command line shell. It is a part of the bash.

Example:

`exit`	terminate the command line shell

`history` (history)

The `history` command displays a list of the executed commands. It is a part of the bash.

Examples:

`history`	print the history list
`!100`	execute the command number `100` in the history list
`history -c`	clear the history list

`poweroff` (power off)

The `poweroff` command brings the operating system down in a safe way. All logged-in users are notified that the system is going down. The `poweroff` command can be performed only by a super user. It is a program in the `/sbin` directory.

Example:

`poweroff`	stop the operating system

`shutdown` (shutdown)

The `shutdown` command brings the operating system down in a safe way. All logged-in users are notified that the system is going down. The `shutdown` command can be performed only by a super user. It is a program in the `/sbin` directory.

Examples:

`shutdown -h now`	stop the operating system immediately
`shutdown -r 10:00`	reboot the operating system at `10:00`

`sleep` (sleep)

The `sleep` command waits for a given amount of seconds. It is a program in the `/bin` directory.

Example:

`sleep 10`	sleep for ten seconds

`sort` (sort)

The `sort` command sorts the lines in a given file alphabetically and numerically. It is a program in the `/usr/bin` directory.

Examples:

`sort list -o sorted`	sort the lines in file `list` and save to the file `sorted`
`sort -r list`	reverse sort and print the lines in file `list`

`su` (substitute user)

The `su` command changes the user identity. It is a program in the `/bin` directory.

Examples:

`su -`	become the root user as if `root` would actually log in (all start-up scripts are processed, see page 29)
`su jekyll`	become the `jekyll` user
`su - hyde`	become the `hyde` user as if the `hyde` user would actually log in

`sudo` (substitute user do)

The `sudo` command allows running the programs with the security privileges of another user (normally a super user). It is a program in the `/usr/bin` directory.

Examples:

`sudo -u hyde ls ~hyde`	list the `hyde` home directory as the `hyde` user
`sudo ./setup`	run the `setup` program in the current directory as a super user

`touch` (touch)

The `touch` command sets the last access date and time of a file to the current date and time. If the file does not exist, it creates an empty file. It is a program in the `/usr/bin` directory.

Example:

`touch data`	set the last access date and time of the `data` file to the current one, create an empty `data` file, if it does not exist

`tty` (teletypewriter)

The `tty` command prints the name of the terminal or console. It is a program in the `/usr/bin` directory.

Example:

`tty`	print the current terminal device

`who` (who)

The `who` command displays who is logged in the system. It is a program in the `/usr/bin` directory.

Example:

`who`	display the information about the logged users

`whoami` (who am I)

The `whoami` command displays the current username. It is a program in the `/usr/bin` directory.

Example:

`whoami`	print the username

1.3.6 Getting help

`man` (manual)

The `man` command provides an in-depth information about a requested command or allows users to search for commands related to a particular keyword [10, 11]. To terminate, press `q`. It is a program in the `/usr/bin` directory.

Examples:

`man ls`	print the manual pages of the `ls` command
`man -k list`	print the manual pages related to the `list` keyword

`whatis` (what is)

The `whatis` command provides a short description of a command. It is a program in the `/usr/bin` directory.

Example:

`whatis ls`	print a short description of the `ls` command

`info` (information)

The `info` command provides information about a requested topic stored in the hypertext files. It is similar to the `man` command, although newer. To terminate, press `q`. It is a program in the `/usr/bin` directory.

Example:

`info ls`	display the information file about the `ls` command

1.4 Redirection

When a command or program is started from a shell, it is given three open files: `stdin` (standard input), `stdout` (standard output) and `stderr` (standard error). Each file has an FD (File Descriptor) number or a handle. The handles for the three standard files are: `0` (`stdin`), `1` (`stdout`) and `2` (`stderr`).

The standard input file pointer points to a device where the input comes from. By default it is a console (or more precisely its input device (e.g. keyboard)) from which the command is started. A standard input of a command can be redirected to an arbitrary file with `<` character (or `0<`, where zero stands for the standard input handle). If a command is run with:

```
command < infile
```

then its standard input file pointer points to the `infile` file instead of the console input device (e.g. keyboard).

Many commands expecting to receive a filename as an argument use a standard input when the filename argument is not given (e.g. `grep 'pattern'`, the argument with a file to be searched is missing, therefore `grep` searches a standard input, or, in other words everything that is typed until `Ctrl-D` (i.e. `EOF`)). Thus, the commands:

```
grep 'pattern' data
grep 'pattern' < data
```

are equivalent. The first searches the `data` file for `pattern`, the second searches the standard input for `pattern`, while the standard input file pointer points to the `data` file.

The standard output file pointer points to a device where the output from a command normally goes to. By default it is a console (or more precisely its output device (e.g. display)) to which the command output is printed. The standard output of a command can be redirected to an arbitrary file with `>` character (or `1>`, where one stands for the standard output handle). If a command is run with:

```
command > outfile
```

then its standard output file pointer points to the `outfile` file instead of the console output device (e.g. display). The command output is saved into `outfile` instead of printed to a standard output device.

The standard error file pointer points to a device where the error output from a command goes to. By default it is the same console (or more precisely its output device (e.g. display)) to which the command output is printed. The command error and regular output are therefore both printed to the same device by default.

As they are still two files, they can be divided by redirection to separate the output from error messages. A standard error of a command can be redirected to an arbitrary file by appending `2>` and a filename. If a command is run with:

```
command 2> errfile
```

then its standard error file pointer points to the `errfile` file instead of the console output device (e.g. display). The command error output is saved into `errfile` instead of printed to an output device.

Linux treats the input (e.g. keyboard), output (e.g. display) and error (e.g. display) devices as files.

Besides the described redirection operators `<` (i.e. `0<`), `>` (i.e. `1>`) and `2>`, there are also `>>` (i.e. `1>>`) and `2>>` that append the command output and error to an existing file instead of creating a new empty file. With a single `>` (`1>` or `2>`) redirection operator, the `&n` can be used as a filename, where `n` stands for a handle. Thus, `2>&1` means redirect the standard error (`2>`) to the file with handle `1` (`&1`). In other words, redirect the standard error to a standard output file. For instance, in the line:

```
command < infile >> outfile 2>&1    ,
```

the command standard input is redirected to `infile`, the standard output is redirected to `outfile`, the existing contents of `outfile` are not deleted, the new contents are appended, the standard error is also redirected to `outfile`.

The output from one command can be assigned as an input into another command by a pipe (`|`) operator. With pipes, the multiple commands can be glued together in a powerful way. For instance, in the line:

```
command1 < infile | command2 | command3 >> outfile    ,
```

the standard input of `command1` is `infile`, the standard input of `command2` is the standard output from `command1`, the standard input of `command3` is the standard output from `command2` and the standard output from `command3` is appended to `outfile`.

1.4.1 Named pipes

As described above, the commands can be glued together using a pipe (`|`) operator. Such pipe is called an anonymous or unnamed pipe. The pipe is used for communication among processes (e.g. commands). It is also called a FIFO (First In First Out) referring to the property that the order of the data going in is the same as the order of the data coming out. With the anonymous pipes, there is one reading and one writing process. That is not the case with the named pipes where more than one reading and more than one writing process may use the pipe. The named pipe is created with the `mkfifo` command. A pipe is visible in the file system as a file. The example given below demonstrates how a named pipe works:

`mkfifo pipe1`	create the `pipe1` named pipe
`ls -l > pipe1`	write a list of files in the current directory to `pipe1`

Manipulate the list of the files in another terminal with:

`grep .dat < pipe1`	read from `pipe1` and print the `.dat` lines

`rm pipe1`	destroy the `pipe1` named pipe

Note that the `ls` command in the first terminal appears to hang. This happens because the other end of the pipe is not yet connected. The `ls` command is suspended (blocked) until the `grep` command opens the pipe in the second terminal.

1.5 Ownership and access rights

Each file, directory, etc., has an owner and associated access rights or permissions. They can be displayed by using the long option with the `ls` command (i.e. `ls -l`), for instance:

```
-rwxr-xr-- 1 maya civilization 123 Dec 20 23:59 doomsday
```

The above line with a long file description has nine fields. The last field is the name of the file (i.e. `doomsday`). The third field defines the user owning the file (i.e. the file `doomsday` is owned by the `maya` user). The fourth field gives the group of the users for which the access rights on the file can be set separately (i.e. access rights on the `doomsday` file can be set separately for the users in the `civilization` group). It is called the group owner of the file. The lists of groups and users belonging to a particular group can be found in `/etc/groups`. A user can be a member of an arbitrary number of groups and has to be a member of at least one group.

The ten characters in the first field (i.e. -rwxr-xr--) are the access rights for the file. The first character indicates the file type which can be:

`-`	file
`d`	directory
`b`	block special file (e.g. `/dev/sda`)
`c`	character special file (e.g. `/dev/console`)
`l`	symbolic link (e.g. `/vmlinuz`)
`p`	pipe (e.g. `/dev/xconsole`; see subsection 1.4.1)
`s`	domain socket (e.g. `/dev/log`; socket is a communication endpoint, e.g. the IP address and port number (see section 2.4) represent an IP socket for exchanging data over TCP/IP (see section 2.1), or the Unix domain socket for exchanging data between processes within the operating system kernel)

The access rights on the file have the following meaning:

`r`	read	permission to read/copy the file
`w`	write	permission to change/rename/delete the file
`x`	execute	permission to run the file as a program

The access rights on the directory have a slightly different meaning:

`r`	read	permission to list the files in the directory
`w`	write	permission to add/rename/delete the files in the directory
`x`	execute	permission to change (e.g. with the `cd` command) into the directory and access files in that directory by name

The nine characters following the type character are divided into three sections defining the permissions for the owner, group and all other users. Each section is three characters long and represents the read, write and execute permissions, respectively (e.g. `-rwxr-xr--` declares that `doomsday` is a file whose owner (i.e. `maya`) has full permissions (i.e. `rwx` - read, write and execute), every user in the `civilization` group has the read and execute permissions (i.e. `r-x`) and all other users have only the read permission (i.e. `r--`)).

A sticky bit (`t`) can appear instead of the execute (`x`) permission for a directory. A file in a sticky directory may only be renamed/deleted by the user having a write permission to the directory and the user is the owner of the file (e.g. `drwxrwtrw-` means that the owner of the directory has full permissions; a member of a group also has full permissions, but can rename/delete only the files he/she owns; all other users have the read and write permissions to the directory meaning that they can rename/delete the files in the directory but cannot change into it). The sticky bit feature is useful for the directories which must be publicly writable but should deny the users to rename/delete each others files (e.g. `/tmp`). The super user can always do everything. The access rights do not apply to the super user.

The fifth field declares the size of the file in bytes (i.e. `doomsday` is `123` bytes long). The following three fields specify the date and time of the last access or modification of the file (i.e. `doomsday` was last accessed on 20^{th} of December at 23:59). If the last access was less than six months ago, then the time is given, otherwise the year is displayed instead of the time.

The second field is the number of hard links to the file. A pointer to the physical location of the file on the disk is called inode (information node). The file system keeps track of where a particular file is stored using the inode pointer structure, where a directory is a special kind of the file containing a list of inodes pointing to the files in the directory. An inode pointer to a file is called a hard link. More than one inode can point to the same physical file (e.g. the directories typically have more hard links, one of them is in the parent directory (`dirname`), one is within itself (`.`) and one is in every child directory (`..`)), thus making the same file to appear in different places under different names. The same but in a slightly different way can be achieved using soft or symbolic links. A soft link is a symbolic path in the directory structure indicating the location of another file.

The hard links cannot cross the file system boundaries, while the soft links can. The soft links are not updated, while the hard links always refer to the source (e.g. a file is physically deleted when the last hard link to it is deleted; on the other hand, if the file is deleted, then all the soft links to it become invalid).

`groups` (groups)

The `groups` command lists the user groups of which the given user is a member. If a username is not specified, the list for the current user is displayed. It is a program in the `/usr/bin` directory.

Examples:

`groups`	list the groups of which the current user is a member
`groups maya`	list the groups of which the `maya` user is a member

`chmod` (change mode)

The `chmod` command sets new access rights on the file. It is a program in the `/bin` directory.

Examples:

`chmod 644 data`	set the `-rw-r--r--` (or binary 0.110.100.100 = `644`) permission to the `data` file
`chmod -R 400 dir`	recursively set the readonly owner permission to the `dir` directory and all its subdirectories
`chmod ug+rw,o-x data`	add the read/write permissions to the owner (user) and group, withdraw the execute permission to other
`chmod a=r data`	set the read permission to all (i.e. `a` is equivalent to `ugo` (user, group, other))

`chown` (change owner)

The `chown` command sets a new user and/or group owner of the file. It is a program in the `/bin` directory.

Examples:

`chown maya doomsday`
: set the `maya` user to be the new owner of the `doomsday` file

`chown maya:civilization doomsday`
: set the `maya` user and `civilization` group to be the new owner of the `doomsday` file

`chown :civilization doomsday`
: set the `civilization` group to be the new owner of the `doomsday` file

`ln` (link)

The `ln` command creates a hard or a soft link to a file. It is a program in the `/bin` directory.

Examples:

`ln ../data copy`	in the current directory create a hard link named `copy` to the `data` file in the parent directory
`ln -s ../data copy`	in the current directory create a symbolic link named `copy` to the `data` file in the parent directory

1.6 Processes and jobs

Linux is a multitasking system. Each task is called a process. A process is a program that is currently being executed. It is identified by a unique process identifier PID (e.g. the `init` program has PID = 1). A process may be in the foreground, in the background, or suspended. In general, the shell does not return with a prompt until the current foreground process finishes executing. If a process takes a long time to finish, then it holds up the terminal. By running such a process in the background, the prompt is returned immediately. Other tasks can be carried out while the original process continues executing in the background.

In Linux, a process can be stopped or suspended by pressing `Ctrl-Z`. By pressing `Ctrl-C`, a process is killed or terminated. The program is a file with an execute permission. The program can be run from the shell prompt in the same way as any command (e.g. the `./program` command will execute the `program` file in the current directory). To run a program in the background, the `&` sign has to be added. For instance, the command:

```
./long_program &
```

will start the `long_program` file in the current directory in the background. The prompt will be returned immediately although `long_program` has not finished executing yet.

The information about a process can be obtained with the `ps` command. The most common process properties are:

`PID`	process identifier number,
`TTY`	control terminal to which the process has the input, output and error,
`TIME`	cumulative CPU time used by the process,
`CMD`	name of the process (i.e. executable program),
`UID` or `USER`	user identifier, the user who owns the process,
`PPID`	parent process identifier number, the parent process that spawned the process,
`C` or `%CPU`	percentage of the CPU time used by the process,
`STIME` or `START`	start time of the process,
`SZ`	process virtual memory usage in kB,
`RSS`	process real memory usage in kB,
`PSR`	processor number to which the process is assigned,
`S` or `STAT`	process status code,
`PRI`	process priority number,
`NI`	process nice value,
`%MEM`	percentage of the memory used by the process, and
`VSZ`	process virtual memory size in kB.

A process has one of the following states:

running	process is either running or ready (waiting to be assigned to CPU) to run (`STAT` = `R`),
waiting	process is waiting for an event (interruptible sleep, `STAT` = `S`) or a resource (non-interruptible sleep, `STAT` = `D`),
stopped	process is suspended (`STAT` = `T`), and
zombie	terminated process, a dead process that for some reason still appears on the list (`STAT` = `Z`).

Additional characters may further describe the process status (`STAT`):

`<`	high-priority process (not nice to others),
`N`	low-priority process (nice to others),
`L`	process with locked memory pages,

`s`	process is a session leader,
`l`	multi-threaded process, and
`+`	foreground process.

`ps` (process status)

The `ps` command lists the current processes with their status information. It is a program in the `/bin` directory.

Examples:

`ps`	list the current running processes
`ps -e`	list all processes
`ps -f`	list the processes in a full format
`ps -F`	list the processes in an extra full format
`ps -ly`	list the processes in a long format
`ps aux`	list all processes (BSD (Berkeley Software Distribution) syntax)
`ps -e | grep program`	print the information on the `program` process

`bg` (background)

The `bg` command continues a suspended job (i.e. process) in the background. It is a part of the bash.

Examples:

`bg`	continue the most recently suspended job in the background
`bg 3`	continue the job number three in the background

`jobs` (jobs)

The `jobs` command lists the user's jobs currently running in the foreground, background or stopped. It is a part of the bash.

Example:

`jobs`	list the current jobs

`fg` (foreground)

The `fg` command continues a suspended job (i.e. process) in the foreground or transfers a job running in the background to the foreground. It is a part of the bash.

Examples:

`fg`	continue the most recently suspended job in the foreground
`fg 3`	continue jthe ob number three in the foreground

`kill` (kill)

The `kill` command sends a signal to a process, usually a request for termination of the process. It is a program in the `/bin` directory. Some most frequently used signals with their codes are:

`TSTP`	20	stop executing (suspend) (`Ctrl-Z`)
`STOP`	19	stop executing (suspend)
`CONT`	18	continue executing if stopped
`ILL`	4	sent to the process at an attempt of executing an unknown instruction
`SEGV`	11	sent to the process at segmentation violation
`INT`	2	request to terminate (`Ctrl-C`)
`TERM`	15	request to terminate (default)
`QUIT`	3	terminate the process and perform a core dump
`ABRT`	6	abort the process
`KILL`	9	terminate immediately

All the signals, except `KILL` and `STOP`, can be intercepted by the process, meaning that a special function can be called when the process (i.e. program) receives the signal. The two exceptions, i.e. `KILL` and `STOP`, are only seen by the kernel. The `TSTP` signal is sent when `Ctrl-Z` is pressed. It is basically the same as `STOP`, although it can be intercepted. The `ILL` and `SEGV` signals are sent by the kernel when a process tries to perform an illegal instruction or makes an invalid memory reference (i.e. segmentation fault). The `INT` signal is sent when `Ctrl-C` is pressed and is similar to `TERM`. `QUIT` performs a core dump which is a current working memory state of a process. The `ABRT` signal is also similar to `TERM`, but cannot be blocked. That means that the process will be unconditionally terminated when the `ABRT` signal handler function returns.

Examples (for a process with PID = `123`):

`kill 123`	terminate (`TERM`) the process
`kill -TSTP 123`	suspend the process (the same as `Ctrl-Z`)
`kill -s CONT 123`	continue the process (the same as the `fg` command)
`kill -9 123`	immediately terminate (`KILL`) the process

`killall` (kill all)

The `killall` command sends a signal to a group of processes in the same way as the `kill` command. It is a program in the `/usr/bin` directory.

Examples:

`killall prog`	terminate (`TERM`) all `prog` processes
`killall -TSTP prog`	suspend all `prog` processes
`killall -s CONT prog`	continue all `prog` processes
`killall -9 prog`	immediately terminate (`KILL`) all `prog` processes

`top` (top)

The `top` command provides an ongoing look at the processor activity in real time. It displays a listing of the most CPU-intensive processes. It is a program in the `/usr/bin` directory. To terminate, press `q`.

Examples:

`top`	display the top processes (the default refresh rate is 1s)
`top -d 0.1`	display the top processes with a 100ms refresh

`lsof` (list the open files)

The `lsof` command displays a list of the open files and the processes that have opened them. Since in Linux everything is either a file or a process, everything that the processes are working with is listed. An open file may be a regular file, directory, device, pipe or socket (see the table on page 21). It is a program in the `/usr/bin` directory.

Examples:

`lsof config.txt`	list the processes using the `config.txt` file
`lsof -c prog`	list the files opened by the processes whose names start with `prog`
`lsof -u hyde`	list the files opened by the processes owned by the `hyde` user
`lsof +p 123`	list the files opened by the process whose PID is `123`
`lsof -i`	list the opened IP sockets
`lsof -i :80`	list the opened IP sockets at port `80` (see section 2.4)
`lsof -U`	list the opened Unix domain sockets
`lsof -i -n -P`	list the opened IP sockets without resolving the hostnames (no DNS, see section 2.7) and port names

`nice` (nice)

The `nice` command invokes a program with a given nice value which modifies the scheduling priority. A higher nice value means a lower priority (the process is nice to others). The range for a nice value goes from -20 (the highest priority) to 19 (the lowest priority). The negative nice values can be set only by a super user. It is a program in the `/usr/bin` directory.

Examples:

`nice -10 prog`	run `prog` with nice = `10`
`nice --10 prog`	run `prog` with nice = `-10`

`strace` (system call/signal trace)

The `strace` command monitors and prints out the running process system calls and received signals. It is a program in the `/usr/bin` directory.

Examples:

`strace prog`
: run the `prog` process and trace its system calls and received signals, write the output of strace to a standard error file

`strace -o out.txt prog`
: run the `prog` process and trace its system calls and received signals, write the output of strace to the `out.txt` file

1.7 Variables and functions

The variables are a set of the name-value pairs that can affect the way the running processes will behave. They create an environment in which a process runs. A variable can be created or changed by:

```
name=value
```

The variables can be used in a command line or in scripts (see section 1.9). They are referenced by putting the `$` sign in front of the name. For instance, to display the value, the `echo` command can be used:

```
echo $name
```

The variables are split into two categories: the environment or global variables and shell or local variables. The shell variables are local to the command line shell in which they are created. They are not inherited by a child process (i.e. a command or program running from the command line shell). The child processes will not be aware of them. When a new variable is created, it is a shell variable by default. On the other hand, a child process inherits from the parent process all the environment variables. To elevate a shell variable into an environment variable, it has to be exported (see page 30). The environment variables are written in the uppercase and the shell variables are written in the lowercase by a non-obligatory convention.

A function is a group of several commands for a later execution using a single name. It can be executed just like a regular command. In contrast to a regular command, a function does not create any new process at execution. A function can be created by:

```
name()
{
  command1
```

```
    command2
       ...
 }
```

A function can be an environment (i.e. global) function or a shell (i.e. local) function. The rules apply in the same way as for the variables.

A default set of the shell and environment variables and functions is defined at shell initialization. Here is a short list of some most common variables:

`HOME`	user's home directory (e.g. `/home/username`)
`HOSTNAME`	host machine name
`OSTYPE`	operating system type (e.g. `linux-gnu`[1])
`PATH`	colon-separated list of directories in which the shell looks for commands
`PPID`	shell parent process identifier
`PS1`	primary prompt string
`PS2`	secondary prompt string (e.g. used when defining a function)
`SHELL`	shell binary (e.g. `/bin/bash`)
`TERM`	user's terminal (e.g. `linux`)
`USER`	username

The start-up scripts can be used to permanently customize or add a variable or a function. They are processed at login each time the shell is invoked. Different shells use different start-up files. Most commonly, the `/etc/profile` and `~/.profile` start-up scripts are executed at login. Which start-up scripts will be processed also depends on how the shell is invoked (e.g. at login, or by executing a shell binary, or to run a script). Some shells also have a logout script (e.g. `~/.bash_logout` for the bash).

`set` (set)

The `set` command displays the names and values of all shell variables and functions. It is a part of the bash.

Example:

`set`	list all variables and functions

`env` (environment)

The `env` command displays the names and values of the environment shell variables and functions. It is a program in the `/usr/bin` directory.

Example:

`env`	list all environment variables and functions

[1] GNU/Linux is a GNU Unix-like operating system (i.e. software collection of applications, libraries and developer tools) with a Linux kernel.

`export` (export)

The `export` command makes a local (i.e. shell) variable or function global (i.e. environment). It is a part of the bash.

Examples:

`export var`	make the `var` shell variable an environment variable
`export var=val`	create a `var` environment variable with value `val`

`unset` (unset)

The `unset` command removes a variable or a function. It is a part of the bash.

Examples:

`unset var`	remove the `var` variable
`unset -f func`	remove the `func` function

1.8 Text file editing

There is a number of different text editors available, but the `vi` (visual) text editor [12] comes with the most versions of Linux. It has powerful features to aid programmers although it is sometimes considered as a terribly user-unfriendly editor.

The `vi` editor uses the full screen, therefore it needs to know what kind of a terminal it is dealing with. The terminal type is defined with the `TERM` variable that should be correctly set end exported. To edit a file with the `vi` editor, type:

```
vi filename
```

If the filename is not given, `vi` will ask for it at an attempt to save the file.

The `vi` editor has two modes: a command and insert. The command mode allows an entry of commands to manipulate the text. The insert mode, on the other hand, inserts the typed characters into the edited file at the cursor position. `vi` starts in the command mode. To change the mode from the command to insert, use the `i` command. To change the mode from insert to command, press `Esc`.

Most commands are one or a few characters long. The most common commands are:

`:w`	save the changes
`:wq`	save the changes and exit
`:q!`	exit without saving the changes
`a`	enter the insert mode after the cursor
`i`	enter the insert mode
`h`	move the cursor left
`j`	move the cursor down
`k`	move the cursor up
`l`	move the cursor right
`x`	delete the character at the cursor
`:d` or `dd`	delete the line

`u`	undo
`J`	join two lines
`y`	copy
`p`	paste
`mc`	set the `c` marker
`‘c`	go to the `c` marker
`"b`	`b` buffer
`/string`	search for `string`
`n`	repeat the last search command
`:s/pattern/string`	replace (substitute) `pattern` with `string`
`:s/pattern/string/g`	replace `pattern` with `string` in the entire file
`&`	repeat the last replace command

To specify `/` character in `string` or `pattern`, use `\/`. For `\` character, use `\\`.

A copy and paste example:

	put the cursor on the first character of the block
`mc`	set the `c` marker
	move the cursor after the last character of the block
`"by‘c`	copy from the current cursor position to the `c` marker into the `b` buffer
	move the cursor to the paste position
`"bp`	paste the `b` buffer

A cut and paste example:

	put the cursor on the first character of the block
`mc`	set the `c` marker
	move the cursor after the last character of the block
`"bd‘c`	cut from the current cursor position to the `c` marker into the `b` buffer
	move the cursor to the paste position
`"bp`	paste the `b` buffer

1.9 Shell scripts

A shell provides an interface between the user and kernel. It is a command interpreter. Besides, it is also a fairly powerful programming language [13, 14]. A shell program is called a script. To run a script, it has to have an execute permission. A detailed explanation of the shell programming exceeds the scope of this script. A glimpse into the shell programming is given with the following example script named `countdown`.

```
#!/bin/bash

check()
{
  if [ "$1" != "" -a "$1" != "-m" -a "$1" != "-h" -a "$1" != "-d" ]
    then
      echo "description: countdown to doomsday (20/12/2012, 24:00)"
      echo "      usage: countdown [-m] [-h] [-d]"
      echo "    options: -m ... to a minute precisely"
      echo "             -h ... to an hour precisely"
```

```
      echo "             -d ... to a day precisely"
      exit
  fi
}

display()
{
  d=$(($1 / 86400))
  h=$(($1 % 86400 / 3600))
  m=$(($1 % 3600 / 60))
  s=$(($1 % 60))
  case "$2" in
     "" )
      printf "\r%4d days, %02d:%02d:%02d to doomsday" $d $h $m $s
      sleep 1
      ;;
    "-m" )
      printf "\r%4d days, %02d:%02d to doomsday" $d $h $m
      sleep $s
      ;;
    "-h" )
      printf "\r%4d days, %02d hours to doomsday" $d $h
      sleep $(($1 % 3600))
      ;;
    "-d" )
      printf "\r%4d days to doomsday" $d
      sleep $(($1 % 86400))
  esac
}

check $1
difference=1
while [ $difference -gt 0 ]
do
  difference=$((1356048000 - `date +%s`))
  display $difference $1
done
printf "\n"
echo "kaboom!"
```

The script counts down to the moment set for 20$^{\text{th}}$ of December 2012 at 24:00. Depending on the specified option, the remaining time is displayed up to a day (`-d`), hour (`-h`), minute (`-m`) or second (default) precisely. The script is run simply by typing its name in the command line (e.g. `./countdown -m`). The first line of the script begins with a shebang or hashbang sequence `#!`. It defines a path to CLI (in our case the bash) which will be used to run the script. The countdown script consists of two functions (`check()` and `display()`) and the main part. The `check()` function checks the specified option, and the `display()` function displays the remaining time and waits until the next change is due. The main part of the script first calls `check()` and then in a loop calculates and displays the remaining time until 20$^{\text{th}}$ of December 2012 at 24:00 (= `1356048000` seconds from 1$^{\text{st}}$ of January 1970 UTC (Universal Time Coordinated)). The script ends with its final message when the moment arrives.

1.10 Programming in C and C++

To accomplish some special task, the user can write a program in the C or C++ programming language [15, 16]. A detailed explanation of the C and C++ programming languages far exceeds the scope of this textbook. When a program is written, it has to be built before it can be run. The very basics on how to build a C program will be demonstrated here by an example. The C version of the shell script example from section 1.9 is presumed to be distributed among the `check.h`, `check.c`, `display.h`, `display.c` and `countdown.c` files as follows:

```
/* check.h */
extern void check(int argc, char **argv);

/* check.c */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void check(int argc, char **argv)
{
 if(argc > 1)
  if(strcmp(argv[1], "-m") && strcmp(argv[1], "-h") &&
     strcmp(argv[1], "-d"))
 {
  printf("description: countdown to doomsday(20/12/2012,24:00)\n");
  printf("      usage: countdown [-m] [-h] [-d]\n");
  printf("    options: -m ... to a minute precisely\n");
  printf("             -h ... to an hour precisely\n");
  printf("             -d ... to a day precisely\n");
  exit(0);
 }
}

/* display.h */
extern void display(int secs, int argc, char **argv);

/* display.c */
#include <stdio.h>
#include <unistd.h>
void display(int secs, int argc, char **argv)
{
  int d = secs / 86400;
  int h = secs % 86400 / 3600;
  int m = secs % 3600 / 60;
  int s = secs % 60;
  if(argc == 1)
  {
    printf("\r%4d days, %02d:%02d:%02d to doomsday", d, h, m, s);
    fflush(stdout);
    sleep(1);
  } else switch(argv[1][1])
  {
  case 'm':
    printf("\r%4d days, %02d:%02d to doomsday", d, h, m);
    fflush(stdout);
```

```
    sleep(s);
    break;
  case 'h':
    printf("\r%4d days, %02d hours to doomsday", d, h);
    fflush(stdout);
    sleep(secs % 3600);
    break;
  case 'd':
    printf("\r%4d days to doomsday", d);
    fflush(stdout);
    sleep(secs % 86400);
  }
}
```

```
/* countdown.c */
#include <time.h>
#include <stdio.h>
#include "check.h"
#include "display.h"
int main(int argc, char *argv[])
{
  int difference;
  check(argc, argv);
  do
  {
    difference = 1356048000 - time(NULL);
    display(difference, argc, argv);
  } while(difference > 0);
  printf("\nkaboom!\n");
  return 0;
}
```

Building the final executable program consists of two steps. First, the source files (with `.c` extension) are compiled to the object files (with `.o` extension). Then the object files are linked into an executable file. In our case, compiling and linking can be done with a few `gcc` [17] (`gcc` stands for the GNU Compiler Collection, also used to mean the GNU C Compiler) calls:

```
gcc -c check.c
gcc -c display.c
gcc -c countdown.c
gcc -o countdown check.o display.o countdown.o
```

The `-c` flag tells `gcc` to compile only. The `-o` flag defines the name of the output file. The subsequent arguments are the input files.
The same can be achieved by only one `gcc` call:

```
gcc -o countdown check.c display.c countdown.c
```

Building the programs consisting of numerous source files can become an awkward task. Therefore, the `make` utility is used. It reads the file named `Makefile` (if not specified otherwise with the `-f` option) and performs the requested action. In general, the makefile is a set of rules of the following form:

```
target: prerequisites
    command
      ...
```

The target is the name of the output file or an action to be carried out. The prerequisites are the input files used to create the target. The subsequent commands are performed by `make` for the requested target. It should be noted that `tab` character has to be at the beginning of every command line. For instance, in our case, `Makefile` can be:

```
# Makefile
all: countdown

check.o: check.c check.h
    gcc -c check.c

display.o: display.c display.h
    gcc -c display.c

countdown.o: countdown.c check.h display.h
    gcc -c countdown.c

countdown: check.o display.o countdown.o
    gcc -o countdown check.o display.o countdown.o

clean:
    rm *.o countdown
```

The `make` command without arguments will follow the rule for the `all` target (default). The prerequisite for `all` is the `countdown`. The prerequisites for the `countdown` are `check.o`, `display.o` and `countdown.o`. Therefore, `make` will first compile the three files (execute `gcc -c ...` commands of the prerequisite targets) and then link them (command `gcc -o ...` of the `countdown` target). The same can be achieved with the `countdown` target:

```
make countdown
```

Cleanup can be performed by the `clean` target:

```
make clean
```

which removes all the object files and final executable.

The `make` utility also automatically determines which pieces of the target need to be recompiled. The pieces, which have not changed since the last compilation are not recompiled.

To run the executable program (e.g. `./countdown`), it has to have an execute permission.

The C++ source code files have the `.cpp` suffix[2]. Regarding the suffix, the `gcc` invokes an appropriate compiler (`gcc` is a collection of compilers). For instance, the C compiler is called on page 34 because of the `.c` suffix. With the `.cpp` suffix, the C++ compiler is invoked. Thus, the C++ source file is compiled to the object file in the same way as the C file:

[2]Suffixes `.C`, `.cc`, `.CPP`, `.c++`, `.cp`, and `.cxx` are also used for the C++ source files.

```
gcc -c source.cpp
```

The `g++` program explicitly invokes the C++ compiler. It actually calls `gcc` with the default language set to C++. All files are treated as the C++ files regardless of their suffix.

```
g++ -c source.cpp
```

When linking the object files into a final executable, `g++` automatically links in the `libstdc++.a` standard C++ runtime library. With `gcc`, the library is not linked by default and has to be manually included.

```
g++ -o executable object1.o object2.o object3.o
gcc -o executable -lstdc++ object1.o object2.o object3.o
```

1.10.1 Source code modifications

To fix a bug in an already released program, a program modification has to be applied. Such a program modification is called a patch. If a program is distributed as a binary executable file, then the patch is also a program modifying or replacing the executable. The end user actually has to run the patch.

When the program source code is available, the final executable file can be compiled and linked by the end user. The patch has to contain the source code modifications, i.e. the textual differences between the original and patched source files. Such a patch can be created by the `diff` command and applied by the `patch` command.

`diff` (difference)

The `diff` command prints the differences between two files or directories. It is a program in the `/usr/bin` directory.

Examples:

`diff org.c new.c`	print the differences between the `org.c` and `new.c` files, i.e. changes required in `org.c` to become `new.c`
`diff -u org.c new.c`	print the differences in a unified format to be used as an input for the `patch` command
`diff -p org.c new.c`	print additional identical lines before and after each difference
`diff orgdir newdir`	run `diff` on files that exist in `orgdir` and `newdir`, report the files that do not exist in both directories
`diff -r orgdir newdir`	diff `orgdir` and `newdir`, recursively descend any matching subdirectories
`diff -N orgdir newdir`	diff files in `orgdir` and `newdir`, the file found in only one directory is compared against an empty file
`diff -uprN orgdir newdir > patch-org.diff`	create the `patch-org.diff` file for patching `orgdir` to `newdir`

`patch` (patch)

The `patch` command updates an original file or directory with a patch file obtained by the `diff` command. A patched version of a file or directory is produced. It is a program in the `/usr/bin` directory.

Examples:

`patch < patch-org.diff`
: apply `patch-org.diff` to files/directories in the current directory; `patch-org.diff` has to be in a unified format

`patch -p1 < patch-org.diff`
: apply the patch, skip the given number of leading slashes in filenames in `patch-org.diff` (e.g. if a file is referenced to as `orgdir/main/source.c`, then with `-p1` the patch will look for `main/source.c` in the current directory)

`patch -p1 -R < patch-org.diff`
: reverse the patch, obtain the original version

1.10.2 Debugging the C and C++ programs

To debug a program, it has to be compiled with a debugging information in it. To include the debugging information, use the `-g` option with `gcc` or `g++`. For instance, an example program from section 1.10 should be compiled with:

```
gcc -g -o countdown check.c display.c countdown.c
```

Now, the obtained executable program (e.g. `./countdown`) can be debugged with `gdb` (GNU Debugger) [18]. A detailed description of the `gdb` features and its commands exceeds the scope of this textbook. Instead, a simple `gdb` session of the `countdown` case is presented for illustration. Invoke `gdb` with the command:

```
gdb countdown
```

The debugger responds with its prompt:

`(gdb) break display.c:9`	set the breakpoint on line `9` in `display.c`
`(gdb) break 13`	set the breakpoint on line `13` in `countdown.c`
`(gdb) run`	start the program, it stops at the first breakpoint (`countdown.c`, line 13)
`(gdb) step`	step into the `display()` function
`(gdb) continue`	continue to the next breakpoint (`display.c`, line 9)
`(gdb) next`	execute one line
`(gbd) print s`	print the value of the `s` variable
`(gdb) ...`	
`(gdb) quit`	close the `gdb` session

1.11 Installing a software package

Linux refers to the family of the Unix-like computer operating systems using the Linux kernel. A particular Linux version is called a distribution. There are many Linux distributions available (e.g. Debian, Fedora (Red Hat), Ubuntu, etc.). Besides, the operating system distributions usually include a large collection of the software applications, such as the software development tools (e.g. `gcc`), interpreted programming languages (e.g. python), typesetting software (e.g. tex), networking applications (e.g. protocol servers (see Chapter 2)), office applications (e.g. open office), etc. The software in the Linux distribution is organized in packages.

The software package management depends on the distribution. For instance, there are various programs and tools available in the Debian distribution [19]. The main package management program is `dpkg` (debian package manager). It can be invoked with various options to unpack, list, install, configure, remove, etc., a particular package. On the next level, there is APT (Advanced Package Tool) consisting of several programs (e.g. `apt-get`). APT can retrieve the specified package from the Internet and call dpkg to install it. The list of the Internet package archives from where the packages can be obtained resides in the `/etc/apt/sources.list` file. There are also other package management tools (e.g. `aptitude`, `synaptic`, `dselect`, etc.). A few basic package management commands for the Debian distribution follow. They have to be executed by a super user.

`dpkg --get-selections`	print a list of all the currently installed packages
`apt-get update`	update the private list of packages with the current available versions
`apt-get upgrade`	upgrade the installed packages to the current available versions
`apt-get dist-upgrade`	upgrade the installed packages to the current available versions, install extra packages required while upgrading the existing ones, remove the obsolete packages
`apt-get install pkg`	download and install the `pkg` package
`apt-get remove pkg`	remove the `pkg` package
`apt-get purge remove pkg`	remove the `pkg` package and its configuration files
`apt-cache search wrd1 wrd2`	find the packages with `wrd1` and `wrd2` words in description
`apt-cache show pkg`	print the information about the `pkg` package
`apt-cache showpkg pkg`	print a detailed information about the `pkg` package (available versions, packages depending on `pkg`)
`apt-cache depends pkg`	print the packages on which the `pkg` package depends

When installing a new package, the list of the already installed packages needs to be updated with the current available versions first. Thus, the `apt-get install pkg` command is normally preceded with the `apt-get update` command.

```
root@host:~# apt-get update
root@host:~# apt-get install pkg
```

Chapter 2

Network

A host is a machine (e.g. computer, printer, etc.) that can be connected to the network [20]. Only two hosts are connected to the network in Fig. 2.1. The connection is simple. The first host can communicate only with the second and vice versa.

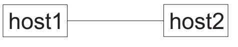

Figure 2.1: Two host network

The principle is extended to the n hosts in Fig. 2.2. In general, each host has $n-1$ connections to every other host in the network. This means that each host should have $n-1$ network interfaces.

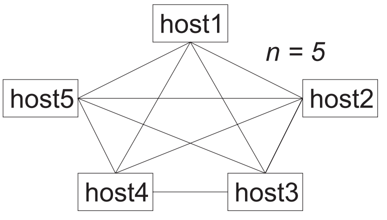

Figure 2.2: n hosts network

To avoid such number of the network interfaces, a hub or a switch is introduced in Fig. 2.3. (Actually, there are three kinds of the network devices: a hub forwards the received packet to all ports, a switch forwards the received packet to a port regarding the destination MAC address (see below) and a router forwards the received packet to a port regarding the destination IP address (see sections 2.1 and 2.2). Often, the hubs and switches are both called the switches, although that is not strictly exact. This textbook also uses the term switch for both.) A switch is a machine connecting all the hosts into a uniform network. Each host has a unique address and only one network interface. It listens to the network to receive packets with its address. If a packet needs to be sent to a particular host, it has to be labeled with the corresponding address. For instance, the network interface Media Access Control (MAC) addresses can be used.

The MAC address is a unique identifier of the network interface. It is assigned by the network interface manufacturer. The MAC address is stored in the interface read-only memory and cannot be changed. Therefore, it is sometimes referred to as a burned-in address. It is a 48-bit number (e.g. 01:23:45:67:89:ab) meaning

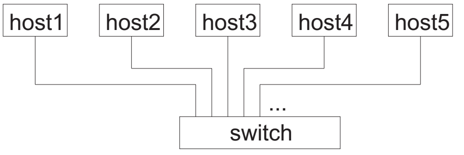

Figure 2.3: Network with a switch

that there are 2^{48} or 281.474.976.710.656 MAC addresses available. It is expected that the 48-bit address space will be exhausted no sooner than by 2100.

Most hosts today are connected into the Ethernet network [21] which has become, over the years, the dominant Local Area Network (LAN) technology. The Ethernet is a broadcast network. This means that the hosts are connected to a network through a single shared medium as shown in Fig. 2.3. Since all the hosts share the same medium, the messages do not have to be routed from the source to the destination. All hosts receive all messages. The MAC addresses are used for host identification. Such configuration raises the media access or collision problem. A collision occurs when two or more hosts try to broadcast at the same time. The Ethernet protocol uses special techniques to prevent collisions (e.g. before broadcasting, a host listens to the media if it is idle). If a collision nevertheless occurs, the Ethernet protocol tries to cope with it as well as possible.

The information is sent around the Ethernet network in data packets called Ethernet frames. The Ethernet frame structure shown in Fig. 2.4 consists of:

- preamble with a start delimiter (seven 10101010 bytes for synchronization followed by a 10101011 delimiter indicating the start of the frame)
- destination MAC address (6 bytes)
- source MAC address (6 bytes),
- data length (2 bytes),
- data (from 0 to 1500 bytes),
- pad (because of the collision detection mechanism, the Ethernet data packet must be at least 64 bytes long (without preamble), if data is less than 46 bytes long, this dummy field is used to compensate), and
- checksum (4 bytes used for error detection and recovery).

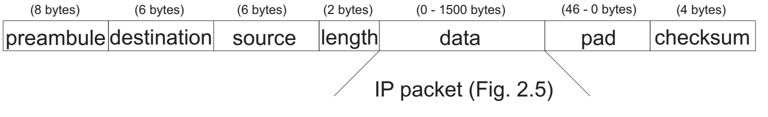

Figure 2.4: Ethernet frame

The Ethernet defines several physical wiring variants, from the coaxial cable to the twisted pair and fiber optic. Today, the Unshielded Twisted Pair (UTP) cables are mostly used with a 100Mbps or 1Gbps (bps - bits per second) bandwidth.

2.1 Internet protocol suite

Above the Ethernet protocol there lays the Internet protocol (IP) [22, 23]. While the Ethernet covers the physical medium and some low-level operation like the message-collision detection, the Internet protocol is responsible for addressing the hosts and routing the packets from a source host to the destination host across one or more IP networks. The IP packet is encapsulated into the data field of the Ethernet frame as shown in Fig. 2.5. It has its own header with the information like the protocol version, packet length, fragmentation information, time to live, transportation protocol, destination and source IP address, etc., to list the most important IP header fields.

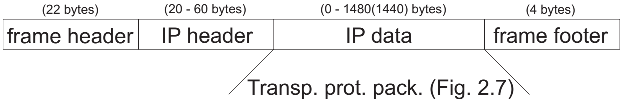

Figure 2.5: Internet protocol packet in Ethernet frame

The hosts using the Internet protocol for communication are labeled with the IP addresses. An IP address is a numerical label assigned to each host attached to the network. It serves as a host identification. The length of the IP address is defined by the IP version. Currently, the IPv4 and IPv6 versions are in use. The IP address is a 32-bit number in IPv4 and a 128-bit number in IPv6. Although IPv4 is obsolete and will be replaced by IPv6 in future, it still carries more than 90% of the worldwide Internet traffic today. This textbook uses the IPv4 examples.

Since the IP address in IPv4 is a 32-bit number, there are 2^{32} (4.294.967.296) unique addresses available. An IP address is canonically represented in a human-readable dot-decimal notation consisting of four decimal numbers, each ranging from 0 to 255, separated by dots (e.g. 172.16.254.1). Each part represents a group of 8 bits (octet) of the address.

If the network shown in Fig. 2.3 uses IP for communication, the switch could theoretically have 2^{32} (4.294.967.296) ports, which is of course not feasible. To solve the issue of the numerous ports, the switches can be arbitrary connected together as depicted in Fig. 2.6.

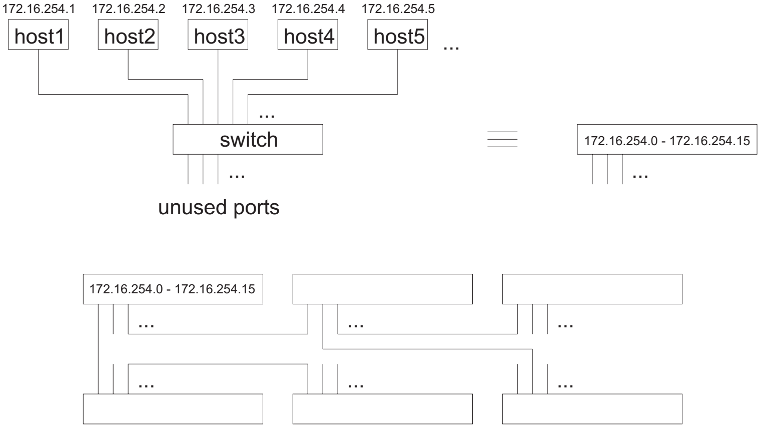

Figure 2.6: Above: a one-switch network, below: a uniform network

The switch does not know about the IP addresses. It forwards the packets regarding the destination MAC address. The Address Resolution Protocol (ARP) is used to find the corresponding MAC address for a given IP address. For example, host One wants to send a packet to host Two with the 172.16.254.2 IP address. In order to send the packet, One needs to know the Two's MAC address. First, One uses its local ARP table to find the MAC address for 172.16.254.2. If the

MAC address is found, One can send the packet. If it is not found, One sends a broadcast ARP message with the ff:ff:ff:ff:ff:ff MAC destination address requesting an answer for 172.16.254.2. Two responds with its MAC address. One inserts an entry for Two into its ARP table for a future use. Two can do the same for One. Thus One obtained the Two's MAC address and the packet can be sent.

The Time To Live (TTL) field in the IP header defines the maximum time the packet is allowed to remain in the Internet system. Every host processing a packet (see section 2.2) must decrease TTL by at least one. If TTL is zero, the packet is destroyed. The undeliverable packet is therefore discarded.

Above the Internet protocol, there lays the transportation protocol. Two transportation protocols are mainly used: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). The transportation protocol packet is encapsulated into the IP data field as shown in Fig. 2.7. Among other information, its header contains the source and destination port number (see section 2.4).

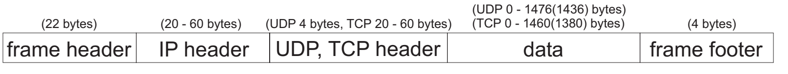

Figure 2.7: Transportation protocol packet in the IP packet in the Ethernet frame

UDP uses a simple transmission model without providing reliability, ordering or data integrity. The sender is not informed back if the packet has been successfully delivered. The UDP packets can arrive out of order, duplicated or go missing without a notice. It is used in applications where dropping a packet is preferable to waiting for it (e.g. Domain Name System (DNS) and Dynamic Host Configuration Protocol (DHCP), see sections 2.7 and 2.9).

On the other hand, TCP provides a reliable and ordered packet delivery. It is designed for accuracy rather than speed. TCP establishes a stream from the sender to the receiver. This means that when a process on one host (i.e. sender) desires to send a chunk of data to a process on another host (i.e. receiver), it just makes a single request to TCP. TCP further handles the IP details by chopping the data into IP packets and restoring the original form at the destination. The entire chunk of data sent in a number of subsequent packets is managed by TCP and is called a stream. TCP automatically takes care of errors like lost, duplicate or out of order packets, etc., by requesting retransmission or reordering, etc. Therefore, TCP guarantees the delivery. It is used by a majority of Internet applications (e.g. HyperText Transfer Protocol (HTTP), Secure SHell (SSH), File Transfer Protocol (FTP), see section 2.4).

The described protocols are often referred to as the Internet protocol suite [24]. The Internet protocol suite is a set of communication protocols used for the Internet. It can be illustrated with four layers as shown in Fig. 2.8. The Internet protocol suite is commonly called TCP/IP. TCP/IP stands for the entire suite although only the names of the two most important protocols are explicitly stated.

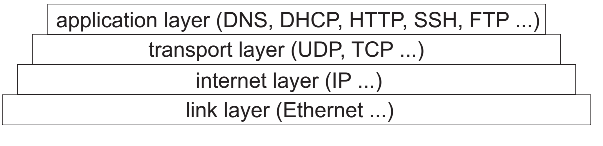

Figure 2.8: Four layers of the Internet protocol suite

2.2 Gateway

In theory, we now have 4.294.967.296 hosts connected to a uniform network. But since each host can hear all the other hosts, only two hosts can communicate over the network at the same time. For this reason, one uniform network is divided into an arbitrary number of subnets connected together through special hosts called the routers or gateways (Fig. 2.9).

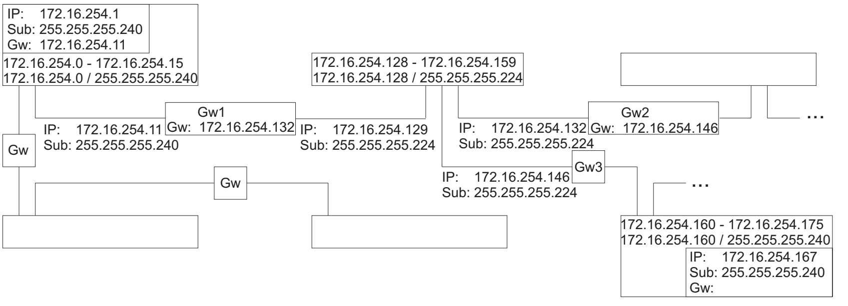

Figure 2.9: Subnets with gateways

A subnet is a uniform network in which hosts within a certain range of the IP addresses reside. It is described with two 32-bit numbers, subnet address and subnet mask. The subnet mask defines which (left side) bits constitute a subnet address.

Example:
Subnet address: 172.16.254.0 =10101100.00010000.11111110.0000 0000
Subnet mask: 255.255.255.240 =11111111.11111111.11111111.1111 0000
The subnet contains 16 IP addresses from 172.16.254.0 to 172.16.254.15.

Besides their IP addresses, the hosts in a subnet also know their subnet mask. The IP address therefore decomposes into two parts, i.e. the subnet address and the host address.

Example:
Host IP address: 172.16.254.1 =10101100.00010000.11111110.0000 0001
Subnet mask: 255.255.255.240 =11111111.11111111.11111111.1111 0000
The host resides in a 172.16.254.0 subnet. Its address is 0.0.0.1.

A gateway is a host with (at least) two network interfaces. It connects (at least) two subnets. Each host, including the gateways, knows its default gateway. The packets sent to the IP addresses within the subnet range will be delivered directly. A packet addressed outside the subnet cannot travel directly to the destination host. It will be delivered to the default gateway for further dispatching to its ultimate destination. A host can act as a gateway only if IP forwarding is enabled (see page 63).
An example (Fig. 2.9): The host 172.16.254.1 sends a packet to host 172.16.254. 167. Since the destination is not within the subnet 172.16.254.0/255.255.255.240 range the packet is sent to the default gateway 172.16.254.11 (Gw1). The destination address is not within the subnet 172.16.254.128/255.255.255.224 range either. It is forwarded to next default gateway 172.16.254.132 (Gw2), and then for the same reason to gateway 172.16.254.146 (Gw3). Since the destination address is

within the subnet 172.16.254.160/255.255.255.240 gateway Gw3 dispatches the packet to its final destination 172.16.254.167.

Since the whole network is divided into subnets, the hosts inside a subnet and the hosts inside another subnet can communicate simultaneously. The hosts inside a subnet cannot hear the hosts in other subnets. Each host can hear only the hosts in its own subnet.

2.3 Routing table

In section 2.2, special hosts called the gateways were introduced. They are used to receive the packets with the destination outside a particular subnet and dispatch them further. Each host knows the IP address of its default gateway. Also, every host, with the exception of the gateways, has only one network interface (Figs. 2.6 and 2.9). To enable a host to use more than one gateway and to have more than one network interface, a routing table is used. If a host has more than one network interface, then it also has to have more than one IP address. One IP address per a network interface has to be assigned.

In fact, every host has a routing table with at least the default gateway listed. Besides the default gateway, the routing table defines where a packet destined to a particular subnet should be dispatched. In other words, it lists the routes to particular network destinations by containing information about the topology of the network immediately around a host. The routing tables are different from one operating system to another, but a routing table always contains the following information:

- subnet,
- gateway, and
- network adapter.

Example: A host with two ethernet network interfaces is shown in Fig. 2.10. The IP addresses of the eth0 and eth1 network interfaces are 212.235.187.70 and 172.16.254.1, respectively.

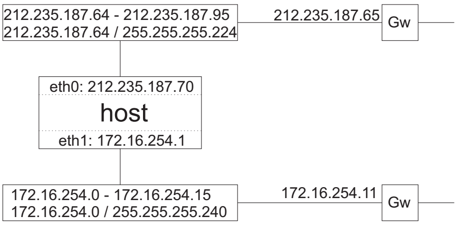

Figure 2.10: Host connected to two subnets with two network interfaces

Since the host has two network interfaces attached to two different subnets, it has a direct access to those two subnets. The routing table lists which network interface and gateway should be used to reach a particular network destination.

As seen from Table 2.1, the host does not have two but three network interfaces: lo0, eth0 and eth1. The lo0 interface is a loopback virtual network interface. For further explanation of the loopback network interface, its 127.0.0.0/255.0.0.0 IP address and subnet, see section 2.8.

Table 2.1 defines that a packet destined to the subnet:

- 127.0.0.0/255.0.0.0 is dispatched directly through the lo0 interface,

subnet	gateway	interface
127.0.0.0/255.0.0.0	not defined	lo0 (127.0.0.1)
172.16.254.0/255.255.255.240	not defined	eth1 (172.16.254.1)
212.235.187.64/255.255.255.224	not defined	eth0 (212.235.187.70)
172.16.254.128/255.255.255.224	172.16.254.11	eth1 (172.16.254.1)
172.16.254.160/255.255.255.240	172.16.254.11	eth1 (172.16.254.1)
0.0.0.0/0.0.0.0	212.235.187.65	eth0 (212.235.187.70)

Table 2.1: Routing table

- 172.16.254.0/255.255.255.240 is dispatched directly through the eth1 interface,
- 212.235.187.64/255.255.255.224 is dispatched directly through the eth0 interface,
- 172.16.254.128/255.255.255.224 is dispatched to the 172.16.254.11 gateway through the eth1 interface,
- 172.16.254.160/255.255.255.240 is dispatched to the 172.16.254.11 gateway through the eth1 interface, and
- 0.0.0.0/0.0.0.0 (all other subnets) is dispatched to the 212.235.187.65 (default) gateway through the eth0 interface.

2.4 Port number

When a packet is received, the host operating system needs to know to which application or process the packet should be delivered. The ports serve that purpose. They complement the IP address. A port is a 16-bit number, thus ranging from 0 to 65.535, which is commonly attached to the IP address with a colon (:) delimiter. Each host process that can receive a packet listens at a unique port. If the host receives a packet addressed to 172.16.254.1:8000, then 172.16.254.1 is obviously the host IP address, while the port number 8000 tells the operating system to deliver the packet to a process listening there.

The processes on a host machine providing a specific network service use the well-known port numbers from 0 to 1023. The well-known port numbers are assigned to services by convention [25]. For instance, the web site server process listens at port 80 (reserved for the HyperText Transfer Protocol (HTTP)). A process or a program providing a service is a daemon. Daemons usually run in the background.

An example: A web browser application on the 172.16.254.2 host connected to the 2049 port wants to receive a web site on the 172.16.254.1 host by HTTP. It sends a request to 172.16.254.1:80 with information that the request came from 172.16.254.2:2049. The web site server on the 172.16.254.1 host, which listens at port 80, gets the request and sends the requested content back to 172.16.254.2:2049.

Some well-known port number / transport layer protocol / service description combinations are listed in Table 2.2.

2.5 Private network

In general, every host using the Internet Protocol for communication has its unique IP address. But if the host is not connected to the global network or Internet,

port	protocol	description
20	TCP	File Transfer Protocol data transfer (FTP)
21	TCP	File Transfer Protocol command (FTP)
22	TCP	Secure SHell (SSH), Secure CoPy (SCP), Secure FTP (SFTP)
23	TCP	Telnet
25	TCP	Simple Mail Transfer Protocol (SMTP)
53	UDP	Domain Name System (DNS)
67	UDP	Dynamic Host Configuration Protocol (DHCP)
68	UDP	Dynamic Host Configuration Protocol (DHCP)
69	UDP	Trivial File Transfer Protocol (TFTP)
80	TCP	HyperText Transfer Protocol (HTTP)
143	TCP	Internet Message Access Protocol (IMAP)
156	TCP	Structured Query Language service (SQL)
443	TCP	HTTP over Secure Sockets Layer (SSL) (HTTPS)

Table 2.2: Some well-known port numbers and assigned services

there is no need for a unique IP address. For instance, the hosts using the Internet Protocol in a private network completely isolated from the Internet can have the IP addresses already used on the Internet. The same IP addresses can be used again and again in different isolated private networks, thus conserving the IP address space.

Three ranges of the IP addresses (Table 2.3) are reserved for the use in private networks [26]. These IP addresses are never used on the Internet. If a packet with a private address appears on the Internet, it is not routed to its destination since there is none.

from	to	number of addresses
10.0.0.0	10.255.255.255	16.777.216
172.16.0.0	172.31.255.255	1.048.576
192.168.0.0	192.168.255.255	65.536

Table 2.3: Reserved private IP addresses

2.5.1 Network Address Translation (NAT)

When a private network needs to be connected to the Internet, NAT has to be used to translate the private IP addresses into one public IP address. A host providing NAT has two network interfaces. One is connected to the private network and the other the with public IP address to the Internet (Fig. 2.11). NAT hides the private IP addresses from the Internet. Concerning the Internet, all the hosts in the private network are hidden behind one public IP address that is the IP address of the host providing NAT.

NAT in general maps the outside port numbers to individual inside private IP addresses. To demonstrate the principle of NAT, the following example will be used.

The web browser application on the 172.16.254.2 host connected to the 2049 port wants to receive a web site on the 212.235.187.72 host by HTTP (Fig. 2.11). The 172.16.254.2 host resides in a private network. It sends a

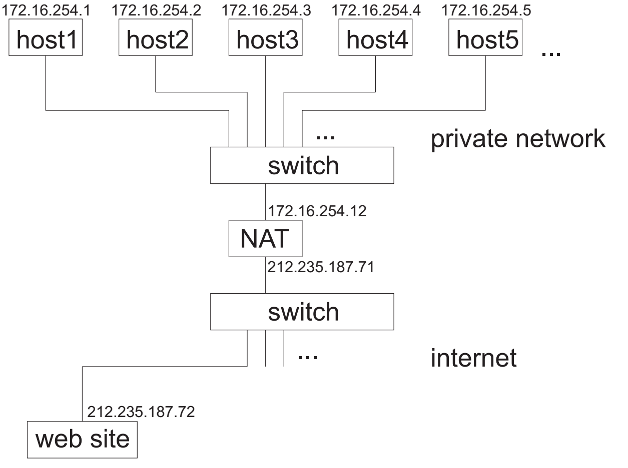

Figure 2.11: Private network with the Internet access through NAT

request to 212.235.187.72:80 with the information that the request came from 172.16.254.2:2049. Since the 212.235.187.72 destination IP address is not inside the private network, the 172.16.254.12 host providing NAT receives the packet. It translates the packet to the Internet by sending it forward under its own public IP address. For instance, the packet addressed to 212.235.187.72:80 and sent from 172.16.254.2:2049 becomes a packet still addressed to 212.235.187.72:80 and sent from 212.235.187.71:9342. The host providing NAT also saves the information that the packet sent from the 9342 port was originally received from 172.16.254.2:2049 in a NAT table.
The web site server on the 212.235.187.72 host, which listens at port 80, gets the request and sends the requested content back to the 212.235.187.71:9342. Therefore, the host providing NAT receives the answer. According to the entry in the NAT table, the NAT host knows that the answer received on port 9342 needs to be further dispatched to its final 172.16.254.2:2049 destination. The configuration of the host providing NAT can be found in subsection 2.14.1.

If the received data has no matching entry in the NAT table, it is ignored and is not dispatched to the private network. Thus the host in the private network cannot receive an unrequested packet. In other words, the unrequested ingoing traffic is blocked. Every entry in the NAT table has a timeout after which it is removed, if not in use. Each new outgoing connection is recorded in the NAT table. If the NAT table is full (e.g. no port available), the new connection is rejected. All new outgoing traffic is blocked until at least one timeouted entry is removed. This phenomenon is also called the NAT overflow.

The described mechanism allows the private network hosts to access the Internet services (e.g. HTTP servers). But on the other hand, they cannot provide services (e.g. they cannot be an HTTP server). They cannot listen at their ports on the Internet. Only the host providing NAT can do that. To overcome this obstacle, the host providing NAT can forward its listening port to a host inside the private network. This is called port forwarding (see subsection 2.14.2). With port forwarding, the host providing NAT can forward its Internet services to a host inside the private network.

2.6 Broadcast IP address

If a host wants to send a packet to all the other hosts in the subnet, the packet should be sent to the broadcast address. The broadcast address addresses all the hosts in the subnet. The subnet broadcast address is an "all-ones" host part of the IP address [27].
An example: to broadcast a packet to the entire 172.16.254.0 subnet with the 255.255.255.240 subnet mask, the 172.16.254.15 broadcast address has to be used (the last four bits of the IP address represent the host part and they are "all-ones").

There is a special definition for the 255.255.255.255 IP broadcast address. It is the broadcast address of the zero network or 0.0.0.0. In IP, the standard zero network stands for this network that is a local network. In general, the packets sent to the 255.255.255.255 IP address are not forwarded outside the subnet by a gateway, if the gateway is not configured otherwise.

2.7 Domain name

The IP addresses are not human-friendly. It is easier to remember the host domain name (e.g. queen.fe.uni-lj.si) than its IP address (e.g. 212.235.187.71). A domain name usually describes a host and its location (e.g. si ... Slovenia, uni-lj ... University of Ljubljana, fe ... Faculty of Electrical Engineering and queen is the machine name). They are read from right to left.

To resolve a domain name into a corresponding IP address, the Domain Name System (DNS) is used [28, 29]. DNS is a hierarchical distributed database and serves as the telephone book for the Internet by translating the human-friendly domain names into the IP addresses. To use DNS, each host has at least one DNS server IP address (apart from the already mentioned its own IP address, subnet mask and default gateway). When a host wants to send a packet to a domain name, it first asks its DNS server for the IP address of the destination. Of course, the DNS server does not have all the IP addresses for all the domain names. It directs the host to the next DNS server serving a particular domain.

For instance, a host wants to send a packet to queen.fe.uni-lj.si. It asks its DNS server for the IP address. The DNS server answers on which DNS server the si domain addresses are. So the host contacts the obtained DNS server which points forward to the next DNS server with the uni-lj.si domain addresses. The host asks the third DNS server and receives a direction to the DNS server with the fe.uni-lj.si domain. Finally, on its fourth query, the host obtains the 212.235.187.71 destination address and sends a packet.

The table of the frequently used domain names is kept locally to avoid using DNS to resolve the same domain name again and again. It resides in the `/etc/hosts` file where pairs of the IP addresses and host domain names are listed.

2.7.1 Uniform Resource Locator (URL)

URL is a reference to a particular resource (e.g. file in some directory on a host) on the network [30]. It defines a protocol to be used for accessing the resource, the host and location of the resource in the host directory structure, etc. Its syntax is:

```
scheme://usename:password@host:port/path?queries#fragment
```

In general, there are eight fields, but all the fields are not specified in most URLs. Individual fields can be omitted. The fields in URL are:

- `scheme` specifies the protocol to be used for accessing the resource (e.g. `http`, `ftp`),
- `username` to be used when authentication is required,
- `password` to be used when authentication is required,
- `host` is the domain name or IP address of the machine with the resource,
- `port` specifies the port number (see section 2.4) where the server process listens,
- `path` specifies where on the host machine the resource can be found,
- `queries` contain data (e.g. parameter names and values separated by ampersands - `par1=val1&par2=val2`) to be passed to the server process, and
- `fragment` specifies a part of the resource.

Examples of URLs can be found on page 55.

2.8 localhost IP address

The standard name for a local machine is `localhost` (`localhost` means this machine) [31]. Its standard IP address is `127.0.0.1`. This means that if one tries to send a packet to `localhost`, the operating system acting as a DNS server resolves the name `localhost` into the `127.0.0.1` IP address. The destination IP address will be routed to the loopback network interface virtually created by an operating system. Thus the local machine will receive the packet bypassing the local network interface hardware.

For instance, a host is an HTTP server and has a loopback network interface configured. A request for the `http://localhost` URL from the web browser installed on the same host will be resolved into `http://127.0.0.1` and routed to the loopback network interface. The host will receive the request and return the home page of the local web site that will be displayed.

Although the `127.0.0.1` IP address is the most commonly used as a `localhost` address, any other IP address from the 127.0.0.0/255.0.0.0 subnet can be used. Therefore, any IP address in the range from 127.0.0.0 to 127.255.255.255 should function in the same manner. The IP addresses in the 127.0.0.0/255.0.0.0 subnet are reserved for the loopback purposes.

2.9 Dynamic Host Configuration Protocol (DHCP)

A host connected to the network has the following IP configuration information:

- IP address,
- subnet mask,
- default gateway IP address,
- DNS server IP address, and
- domain name (stored in DNS).

The host has to be configured before it can communicate with other hosts on the network. To avoid manual configuration of each host, DHCP can be used [32]. With DHCP, a host can be configured automatically without administrator intervention. The use of DHCP also prevents two hosts to be accidentally configured with the same IP address.

The hosts configured by DHCP are the DHCP clients. A host providing a DHCP service is called a DHCP server. A DHCP server manages a pool of the client IP configuration parameters, such as the IP address, subnet mask, default gateway, DNS sever and domain name. It keeps a track of the allocated configurations and their leases. A lease is a length of time the configuration allocation is valid.

At boot, a DHCP client broadcasts a request over its subnet to discover an available DHCP server. The request is sent to the "all-ones" subnet broadcast address or to the general 255.255.255.255 broadcast address. In general, if the gateways connected to the subnet are not configured otherwise, the request can be heard only in DHCP client's subnet.

When a DHCP server receives a request from a DHCP client, it reserves an IP configuration (IP address, etc.) for the client. The DHCP server broadcasts its offer back to the client. Since the client does not have a valid IP address yet, the DHCP server offer is again sent to the broadcast address. The offer contains the client MAC address, the IP address that the server is offering, the subnet mask, the lease duration, and the IP address of the DHCP server making the offer, etc.

The DHCP client receives the server offer and broadcasts back an acceptance packet. If multiple DHCP offers are received from several DHCP servers, the client accepts only one. The servers will be informed whose offer is accepted by receiving the client broadcasted acceptance. By broadcasting, the client informs all the servers about its decision with a single packet. The rejected servers will withdraw their offers and return the offered IP configuration (IP address, etc.) to the pool of available configurations.

Finally, the DHCP server receives the acceptance from the client and broadcasts back the acknowledgement. The IP configuration process is completed. The DHCP client configures its network interface with the obtained IP configuration. An IP configuration is valid only for a predefined amount of time, a lease. Once half of the lease interval expires, the client starts a renewal.

The DHCP server software can be found in the `isc-dhcp-server` package of the Debian Linux distribution. The server binary (e.g. `dhcpd`) is installed in the `/usr/sbin` directory. If not running, it can be started with the following command issued as a super user:

```
/etc/init.d/isc-dhcp-server start
```

The messages are logged into the `/var/log/syslog` file. The DHCP server configuration is set in the `/etc/dhcp/dhcpd.conf` file. For instance, an entry:

```
subnet 192.168.56.0 netmask 255.255.255.0 {
  host imx27 {
    hardware ethernet 00:50:c2:5e:00:9a;
    fixed-address 192.168.56.100;
    option host-name "imx27";
  }
}
```

defines that the `192.168.56.100` IP address will be assigned to the host named `imx27` with the Ethernet network interface `00:50:c2:5e:00:9a` MAC address. The `imx27` host resides in the `192.168.56.0/255.255.255.0` subnet.

2.10 Basic network-related commands

`hostname` (hostname)

The `hostname` command displays or sets the host machine name. It is a program in the `/bin` directory.

Examples:

`hostname`	display the host machine name
`hostname queen`	set the host machine name to the queen (can be performed only by a super user)

The change of the host machine name made by the `hostname` command is temporary. The name used at the system boot resides in the `/etc/hostname`. To permanently set the host machine name, the `/etc/hostname` file has to be edited.

`ifdown` (interface down)

The `ifdown` command takes the network interface down. It can be executed only by a super user. It is a program in the `/sbin` directory.

Example:

`ifdown eth0`	take the `eth0` interface down

`ifup` (interface up)

The `ifup` command brings the network interface up regarding the configuration in the `/etc/network/interfaces`. It can be executed only by a super user. It is a program in the `/sbin` directory.

Example:

`ifup eth0`	bring the `eth0` interface up

`ifconfig` (interface configuration)

The `ifconfig` command is a system administration utility for setting and viewing the network interface parameters to be executed only by a super user. It is a program in the `/sbin` directory.

Examples:

`ifconfig`	display the status of the currently active interfaces
`ifconfig eth0`	display the status of the `eth0` interface
`ifconfig eth0 down`	set the `eth0` interface inactive
`ifconfig -a`	display the status of all the interfaces

	(inactive included)
`ifconfig eth0 up`	set the `eth0` interface active
`ifconfig eth0 172.16.254.1 netmask 255.255.255.240`	temporary set the IP address and subnet mask for the `eth0` interface

The configuration changes made with the `ifconfig` command are temporary. The network interface configurations used at the system boot reside in the `/etc/network/interfaces`. To permanently set the network interface configuration, the `/etc/network/interfaces` file has to be edited. The network interface configuration changes made in `/etc/network/interfaces` can take effect without booting by using the `ifdown` command followed by the `ifup` command on the changed interface.

`route` (route)

The `route` command displays and manipulates the routing table (see section 2.3). It can be executed only by a super user. The route command is a program in the `/sbin` directory.

Examples:

`route`	display the routing table using the host names obtained by the DNS (DNS servers listed in the `/etc/resolv.conf`) or `/etc/hosts` file
`route -n`	display the routing table using the IP addresses
`route add -net 172.16.254.0 netmask 255.255.255.240 eth0`	add a route to the `172.16.254.0/255.255.255.240` subnet through the `eth0` interface
`route del -net 172.16.254.0 netmask 255.255.255.240`	delete the route to the `172.16.254.0/255.255.255.240` subnet
`route add -net 172.16.254.128 netmask 255.255.255.224 gw 172.16.254.11 eth0`	add a route to the `172.16.254.128/255.255.255.224` subnet via the `172.16.254.11` gateway through the `eth0` interface
`route del -net 172.16.254.128 netmask 255.255.255.224`	delete the route to the `172.16.254.128/255.255.255.224` subnet
`route add -net default gw 172.16.254.12 eth0`	add a default route to the unlisted subnets via the `172.16.254.12` gateway through the `eth0` interface
`route del -net default`	delete the default route to the unlisted subnets
`route add -host 172.16.254.162 eth1`	add a route to the `172.16.254.162` host through the `eth1` interface
`route del -host 172.16.254.162`	delete the route to the `172.16.254.162` host
`route add -net 172.16.254.176 netmask 255.255.255.240 reject`	add a blocking route to mask-out the `172.16.254.176/255.255.255.240` subnet

The routing table changes made with the `route` command are temporary. They will not be re-established at the next system boot. After the system boot, the routing table is built with respect to the `/etc/network/interfaces` file. Special static routes can be added. The `eth1` interface configuration in the `/etc/network/interfaces` is for instance:

```
iface eth1 inet static
  address 172.16.254.1
  network 172.16.254.0
  netmask 255.255.255.240
  up route add -net 172.16.254.128 netmask 255.255.255.224
                                                    gw 172.16.254.11
  down route del -net 172.16.254.128 netmask 255.255.255.224
```

The configuration specifies the interface IP address and subnet which automatically adds a route to the `172.16.254.0/255.255.255.240` subnet through `eth1`. An additional static route through `eth1` to the `172.16.254.128/255.255.255.224` subnet via the `172.16.254.11` gateway is added with an `up` line. The lines `up` and `down` are executed when the interface is brought up and token down, respectively. To avoid booting, the `ifdown` and `ifup` commands can be used.

`ping` (ping)

The `ping` command checks if there is a network connection to another host. It is a program in the `/bin` directory.

Examples:

`ping queen.fe.uni-lj.si`	get response from the `queen.fe.uni-lj.si` host, to terminate, press `Ctrl-C`
`ping -c 5 212.235.187.71`	ping the `212.235.187.71` host five times

`telnet` (telecommunications network)

The `telnet` command enables the text terminal connection to a remote host. All the data transfers, including the usernames and passwords, are in a clear text. Therefore, the `telnet` session is insecure and considered obsolete. Use `ssh` instead. The `telnet` command is a program in the `/usr/bin` directory.

Example:

`telnet queen.fe.uni-lj.si`
  open the `telnet` session to the `queen.fe.uni-lj.si` host

`ssh` (secure shell)

The `ssh` command opens a command line shell at a remote host. The data traffic is encrypted by a symmetric encryption. It is a program in the `/usr/bin` directory.

Examples (see also pages 65 and 73):

`ssh freddie@queen.fe.uni-lj.si`
: open the command line shell as the `freddie` user at the `queen.fe.uni-lj.si` host

`ssh queen.fe.uni-lj.si`
: open the command line shell with the client machine username at the `queen.fe.uni-lj.si` host

`ssh freddie@212.235.187.71`
: open the command line shell as the `freddie` user at the host with the `212.235.187.71` IP address

The SSH network protocol is used to secure the communication between the client and the server (remote) host over an insecure network [33]. On the client side, an SSH client process (e.g. `/usr/bin/ssh`) is required. On the server side, the SSH server process (e.g. `/usr/sbin/sshd`) continually listens for the clients requesting an SSH connection. Before an SSH session begins, an asymmetric encryption (public keys are exchanged, the data is encrypted with a public key, decrypted with a private key) is used to obtain a symmetric encryption key (the data is encrypted and decrypted with the same key). At the first connection to a remote host, the SSH protocol asks if the host should be added to the list of the known hosts residing in the client `~/.ssh/known_hosts` file. The host keys in the known host list are used in further sessions for host validation to avoid the man-in-the-middle attacks (i.e. prevent logging into a fake host used to sniff the session). If the `~/.ssh/known_hosts` file is removed, the next connection to each remote host is considered as the first. The SSH system-wide client and server configuration, including the encryption keys, can be found in the `/etc/ssh` directory. An individual user SSH client configuration adjusting the system-wide settings to a particular user can be found in the `~/.ssh` directory.

`scp` (secure copy)

The `scp` command copies the files between the hosts using the SSH connection. It is a program in the `/usr/bin` directory.

Examples:

`scp *.mp3 freddie@queen.fe.uni-lj.si:/home/freddie/mp3`
: copy the `.mp3` files in the current directory into the `/home/freddie/mp3` directory on the `queen.fe.uni-lj.si` remote host using the `freddie` user account

`scp -r freddie@212.235.187.71:/home/freddie/mp3 ./mp3`
: recursively copy all the files and directories in the `/home/freddie/mp3` directory residing on the `212.235.187.71` remote host to the `mp3` directory in the current directory on the local host using the `freddie` user account

`ftp` (file transfer program)

The `ftp` command enables transferring the files to and from a remote host using FTP. All data transfers, including usernames and passwords, are in a clear text.

Therefore, the FTP session is insecure and considered obsolete. Use `sftp` instead. The `ftp` command is a program in the `/usr/bin` directory. After logging into a remote host, the `ftp` command opens its own CLI, where the FTP commands can be used. To terminate, use the `quit` command.

Examples:

`ftp queen.fe.uni-lj.si`	log the into `queen.fe.uni-lj.si` remote host and open CLI
`ftp 212.235.187.71`	log into the `212.235.187.71` remote host and open CLI

`sftp` (secure file transfer program)

The `sftp` command enables transferring the files to and from a remote host using SFTP which is an extension of the SSH protocol. It is a program in the `/usr/bin` directory. After logging into a remote host, the `sftp` command opens its own CLI, where the SFTP commands can be used. To terminate, use the `quit` command.

Examples:

`sftp freddie@queen.fe.uni-lj.si`
: log into the `queen.fe.uni-lj.si` remote host as the `freddie` user and open CLI

`sftp queen.fe.uni-lj.si`
: log into the `queen.fe.uni-lj.si` remote host with the client machine username and open CLI

`sftp freddie@212.235.187.71`
: log into the `212.235.187.71` remote host as the `freddie` user and open CLI

SFTP is used to secure the file transfer between the client and server (remote) host over an insecure network. On the client side, an SFTP client process (e.g. `/usr/bin/sftp`) is required. There is no SFTP server process on the server side. The SSH server process (e.g. `/usr/sbin/sshd`) continually listens for the clients requesting an SSH connection which includes the SFTP requests.

`wget` (world wide web get)

The `wget` command is a client program retrieving the content from a server using HTTP, HTTPS or FTP. On the server side, appropriate web server processes serving the HTTP, HTTPS and FTP requests are required. The `wget` command is a program in the `/usr/bin` directory.

Examples:

`wget http://queen.fe.uni-lj.si/~freddie/data.tar.gz`
: download the `data.tar.gz` file from the World Wide Web home directory of the `freddie` user (e.g. `/home/freddie/public_html`) at `queen.fe.uni-lj.si` using HTTP

`wget -o log.txt http://212.235.187.71`

download the title page (e.g. `/var/www/index.html`) at `212.235.187.71` using HTTP, save the progress messages to `log.txt`

`wget -k -r -l 3 http://queen.fe.uni-lj.si/~freddie`
: download a complete web site of the `freddie` user at `queen.fe.uni-lj.si` up to three levels deep using HTTP, convert the links to point to the downloaded files to enable offline viewing

`wget ftp://queen.fe.uni-lj.si/pub/data.tar.gz`
: download the `data.tar.gz` file from a `pub` subdirectory (e.g. `/var/ftp/pub`) at `queen.fe.uni-lj.si` as an `anonymous` user using FTP

`wget ftp://freddie:vocal@212.235.187.71/pub/data.tar.gz`
: download the `data.tar.gz` file from a `pub` subdirectory (e.g. `/home/freddie/pub`) at `212.235.187.71` as the `freddie` user with the `vocal` password using FTP

The Apache is the most widely used HTTP server program on the Linux-powered servers. Its main title page is `index.html` in the `/var/www` directory. The paths specified in URL (see subsection 2.7.1) are relative to the `/var/www` HTTP home directory. With an appropriate HTTP server configuration, each user can have his/hers own HTTP home directory. If so, and if the user is specified in URL, then the path is relative to the user's HTTP home directory (e.g. `/home/username/public_html`). The configuration of the Apache HTTP server resides in the `/etc/apache2` directory. Similar applies to the FTP servers (e.g. `/var/ftp` is an FTP home directory for the `anonymous` user, and the user's home directory (e.g. `/home/username`) is used when the user is specified).

2.11 Network File System (NFS)

NFS is a distributed file system protocol [34] allowing a transparent network access to the files on the NFS server host. The files on the server are accessed from a client as they are local (see Fig. 2.12). On the server side, the NFS server process (e.g. `nfsd` kernel module) continually listens for the client requests. The NFS client software can be found in the `nfs-common` and `portmap` packages of the Debian Linux distribution, while the NFS server also requires the `nfs-kernel-server` package.

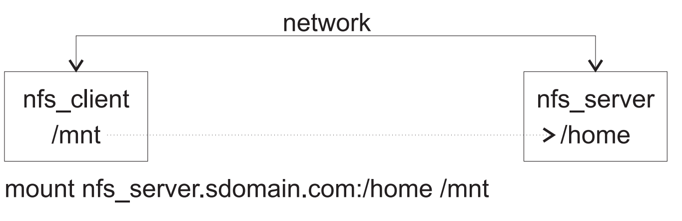

Figure 2.12: Network file system

The server directories exportable to the NFS clients are listed in the `/etc/exports` file. Each line begins with an absolute path to the directory to be exported, followed by a list of clients accessing the directory, e.g.:

```
/home nfs_client.cdomain.com(rw,no_root_squash,sync)
                                    172.16.254.0/255.255.255.240(ro)
```

The above line exports the /home directory and grants a read-write access to the nfs_client host (instead of the domain name, the IP address can also be used), and a read-only access to all hosts in the 172.16.254.0/255.255.255.240 subnet. In this example, the no_root_squash and sync options are also used. The no_root_squash option means that the root on the client is also considered the root on the server. The sync option specifies that the NFS server replies to requests only after the changes have been committed (e.g. written to the disk). Thus, the data cannot be lost or corrupted at an eventual server crash. The /etc/exports file can be edited only by a super user. The changes in /etc/exports become effective by the command:

```
root@nfs_server:~# exportfs -r
```

A super user on the client host can mount the exported directory by:

```
root@queen:~# mount nfs_server.sdomain.com:/home /mnt
```

The IP address can be used instead of the NFS server domain name.

2.12 HyperText Markup Language (HTML)

HTML is the predominant markup language for the web pages where the documents written in HTML are the basic building blocks [35]. An HTML document consists of elements holding the web site content. An element starts and ends with a pair of tags enclosed in angle brackets. A detailed description of HTML far exceeds the scope of this textbook. For demonstration, an example of a default HTML document (e.g. /var/www/index.html) on the web site server follows:

```
<html>
  <body>
    <h1>
      It works!
    </h1>
    <p>
      This page is the default web page for this server.
    </p>
    <p>
      The web server software is running but no content has been
      added yet.
    </p>
  </body>
</html>
```

2.13 Programming in the JavaScript and PHP Hypertext Preprocessor (PHP) languages

Both scripting languages, i.e. JavaScript [36] and PHP [37], are used for creating dynamic web pages, i.e. the HTML documents. The main difference between the two is in who interprets the code. When JavaScript is used, the server sends the entire dynamic HTML document to the client, i.e. the web browser application, which runs the JavaScript code to obtain the final HTML document. JavaScript is the client-side scripting language. On the other hand, the PHP code in a dynamic HTML document runs on the server side, i.e. the PHP module of the web site server process. The obtained final HTML document is then sent to the client. Therefore, the PHP is a server-side scripting language.

A detailed description of the two scripting languages far exceeds the scope of this textbook. A simple calculator example implemented in JavaScript and PHP is given in Fig. 2.13. There are two input fields for the operands and operation selection drop down menu. The result is displayed when the = button is pressed.

Figure 2.13: Simple calculator example of a dynamic HTML document

The first is the JavaScript code. The final HTML document actually consists of one form element. The = button submits the input values to the same URL (see subsection 2.7.1) using the `get` method. That means that the parameter name and value pairs are passed in the queries field of URL (e.g. `http://queen.fe.uni-lj.si/calculator.html?first=123&operation=add&second=456`). To extract a particular parameter value from the `queries` string the `getParam()` function is used.

```
<html>
 <head>
  <script language="javascript">
   function getParam(name)
   {
    var i, queries;
    if(window.location.search.length == 0) return "";
    queries = window.location.search.substring(1).split('&');
    for(i = 0; i < queries.length; i++)
     if(queries[i].indexOf(name) == 0)
      return queries[i].split('=')[1];
    return "";
   }
  </script>
 </head>
 <body>
  <script language="javascript">
   var address = window.location.href.split('?')[0].split('#')[0];
   var first = getParam("first") * 1;
   var operation = getParam("operation");
   var second = getParam("second") * 1;
   document.write("<form action='" + address + "' method='get'>");
   document.write("<input type='text' name='first' value='" +
                                                    first + "' />");
   document.write(" <select name='operation'>");
   if(operation == "sub")
   {
    document.write("<option value='add'>+</option>");
    document.write("<option value='sub' selected>-</option>");
   } else
   {
    document.write("<option value='add' selected>+</option>");
    document.write("<option value='sub'>-</option>");
   }
```

```
   document.write("</select> ");
   document.write("<input type='text' name='second' value='" +
                                          second + "' /> ");
   document.write("<input type='submit' value='=' /> ");
   if(operation == "add") document.write(first + second);
   else document.write(first - second);
   document.write("</form>");
  </script>
 </body>
</html>
```

The same can be achieved on the server side with an equivalent code in PHP:

```
<html>
 <head>
  <script language="php">
   function getParam($name)
   {
    return $_GET[$name];
   }
  </script>
 </head>
 <body>
  <script language="php">
   $address = "http://" . $_SERVER['SERVER_NAME'] .
                                          $_SERVER['PHP_SELF'];
   $first = getParam("first");
   $operation = getParam("operation");
   $second = getParam("second");
   echo "<form action='" . $address . "' method='get'>";
   echo "<input type='text' name='first' value='" . $first . "' />";
   echo " <select name='operation'>";
   if($operation == "sub")
   {
    echo "<option value='add'>+</option>";
    echo "<option value='sub' selected>-</option>";
   } else
   {
    echo "<option value='add' selected>+</option>";
    echo "<option value='sub'>-</option>";
   }
   echo "</select> ";
   echo "<input type='text' name='second' value='" . $second .
                                                          "' />";
   echo " <input type='submit' value='=' /> ";
   if($operation == "add") echo $first + $second;
   else echo $first - $second;
   echo "</form>";
  </script>
 </body>
</html>
```

In both cases, i.e. JavaScript and PHP, the `get` method is used to pass the parameters to the server. The name/value pairs are sent as `queries` in URL. A large parameter set, i.e. hundreds of characters, can cause problems because of the extremely long URL. Another problem are eventual non-ASCII characters in the parameter names or values which should not appear in URL. In such cases, the `post` method has to be used. The parameters are not part of URL with the `post` method. They are sent within the body of the HTML request. The posted parameters are received on the sever side meaning that they cannot be obtained by JavaScript. Note that both methods, i.e. `get` and `post`, send the parameters as a plain text, thus unsecured. To use the `post` method, only two lines in the PHP example have to be modified. Instead of the array of the `$_GET` queries, the array of the `$_POST` posted parameters has to be used, and the `form` element has to use the `post` method:

```
           ...
line 6:   return $_POST[$name];
           ...
line 16:  echo "<form action='" . $address . "' method='post'>";
           ...
```

The PHP scripts are enclosed in the `<script language="php"> ... </script>` element. The shorter `<?php ... ?>` form is more commonly used.

2.14 Firewall

A firewall represents a system for the network traffic control. It can be implemented as a hardware device or a piece of software. A firewall is configured by a set of rules upon which an individual packet is permitted or denied its further route to its final destination. By denying the packets not meeting the specified criteria, the firewall prevents an unauthorized access. It can be used to protect an individual host or an entire subnet. In the latter case, it is positioned on a key machine connecting different subnets (e.g. gateway, host providing NAT).

The firewall is a part of the Linux kernel. The rules upon which the firewall operates are managed by the `iptables` command [38]. It is a program in the `/sbin` directory. The `iptables` command can be performed only by a super user. The firewall management is slightly different form distribution to distribution. A few basics for the Debian distribution follow.

The firewall configuration is organized in tables containing chains which themselves contain rules. The most important are the filter and nat tables. The filter is the default table with three predefined chains:
- `INPUT` for the arriving packets to a local host,
- `OUTPUT` for the departing packets from alocal host, and
- `FORWARD` for the arriving packets not destined to a local host and to be only routed through.

The nat table provides the NAT rules in the following three predefined chains:
- `PREROUTING` for packet translation at arrival before routing,
- `POSTROUTING` for packet translation at departure after routing, and
- `OUTPUT` for translation of the locally generated packets.

There are also the mangle (packet alteration) and raw (exceptions) tables which exceed the scope of this textbook. None of the chains in none of the tables contains no rules by default. Their default policy is `ACCEPT` meaning that a packet matching no rule is accepted. An individual table can be listed with the `-L` option:

```
root@host:~# iptables -t nat -L
```

```
Chain PREROUTING  (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING  (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT  (policy ACCEPT)
target     prot opt source               destination
```

The default firewall status is sometimes referred to as off. In fact, the firewall cannot be turned off. By default it just accepts all packets and denies none. A packet is tested against the rules in a chain orderly until it is accepted, dropped or some other final target is specified. If a rule is not matched, testing proceeds with the next rule. If no rule is matched, the chain default policy is applied. A few examples of setting the rules follow.

Deleting the rules. Delete all rules in all chains in the filter table:

```
iptables -t filter -F
```

Setting the default policy. Set the default policy to `DROP` for the `INPUT` chain in the filter table. A packet matching no rule in the `INPUT` chain is dropped.

```
iptables -t filter -P INPUT DROP
```

Block a specific IP address. Add a rule to drop all packets received from the `172.16.254.1` host.

```
iptables -t filter -A INPUT -s 172.16.254.1 -j DROP
```

Allow connections to the SSH server on a local host. The SSH server listens on port 22 and uses TCP on the transport layer. Connections are allowed on the `eth0` network interface.

```
iptables -t filter -A INPUT -i eth0 -p tcp --dport 22 -m state
                                  --state NEW,ESTABLISHED -j ACCEPT
iptables -t filter -A OUTPUT -o eth0 -p tcp --sport 22 -m state
                                      --state ESTABLISHED -j ACCEPT
```

Allow connections to the SSH server only from the `172.16.254.0/255.255.255.240` subnet.

```
iptables -t filter -A INPUT -i eth0 -p tcp -s
           172.16.254.0/255.255.255.240 --dport 22 -m state --state
                                        NEW,ESTABLISHED -j ACCEPT
iptables -t filter -A OUTPUT -o eth0 -p tcp --sport 22 -m state
                                      --state ESTABLISHED -j ACCEPT
```

Allow connections to the SSH, HTTP and HTTPS servers on a local host. The SSH server listens on port 22, HTTP on port 80 and HTTPS on port 443.

```
iptables -t filter -A INPUT -i eth0 -p tcp -m multiport --dports
               22,80,443 -m state --state NEW,ESTABLISHED -j ACCEPT
```

```
iptables -t filter -A OUTPUT -o eth0 -p tcp -m multiport --sports
                    22,80,443 -m state --state ESTABLISHED -j ACCEPT
```

Allow connections to the SSH server on a remote host.

```
iptables -t filter -A OUTPUT -o eth0 -p tcp --dport 22 -m state
                                   --state NEW,ESTABLISHED -j ACCEPT
iptables -t filter -A INPUT -i eth0 -p tcp --sport 22 -m state
                                       --state ESTABLISHED -j ACCEPT
```

Allow ping from a remote to a local host. Ping uses the ICMP (Internet Control Message Protocol) internet layer protocol.

```
iptables -t filter -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -t filter -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

Allow ping from a local to a remote host.

```
iptables -t filter -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -t filter -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

Allow access to a localhost through a loopback network interface.

```
iptables -t filter -A INPUT -i lo -j ACCEPT
iptables -t filter -A OUTPUT -o lo -j ACCEPT
```

Allow one subnet to another communication. The packets received from a subnet at the `eth0` network interface are allowed to continue their route to another subnet at the `eth1` network interface.

```
iptables -t filter -A FORWARD -i eth0 -o eth1 -j ACCEPT
```

Allow connections to the DNS server on a remote host. The DNS server listens on port `53` and uses UDP on the transport layer. Connections are allowed on the `eth0` network interface.

```
iptables -t filter -A OUTPUT -p udp -o eth0 --dport 53 -j ACCEPT
iptables -t filter -A INPUT -p udp -i eth0 --sport 53 -j ACCEPT
```

Log the packet information. The rule is always matched. Therefore, a log record is made for all the packets tested against the rule. Logging is specified by the LOG target which is not final. Thus, a packet is neither accepted nor dropped and testing proceeds with the next rule. If this rule is the first rule in the `INPUT` chain, all the received packets are logged. The system logging is provided by the `syslog` daemon (e.g. `/usr/sbin/rsyslogd`). The records are written into the kernel log file `/var/log/kern.log`. The log records are prefixed for easier recognition.

```
iptables -t filter -A INPUT -j LOG --log-prefix "IPTables received
                                             packet (INPUT chain): "
```

Thr current set of rules can be saved with the `iptables-save` command. It is a program in the `/sbin` directory.

```
iptables-save > /etc/iptables.up.rules
```

Such saved set of rules can be activated with the `iptables-restore` command. It is a program in the `/sbin` directory.

```
iptables-restore < /etc/iptables.up.rules
```

There are no rules at the boot by default. The firewall is turned off. To activate a set of rules at the boot the `iptables-restore` command has to be issued in the `/etc/network/if-pre-up.d/iptables` file, for instance:

```
#!/bin/bash
/sbin/iptables-restore < /etc/iptables.up.rules
```

2.14.1 NAT configuration

The NAT service can be configured by a rule in the `POSTROUTING` chain of the nat table. The command below adds a rule to set up masquerading for all the packets leaving at the `eth1` network interface. The `MASQUERADE` target actually provides the NAT service, i.e. assigns a port to the packet original source (see subsection 2.5.1). The case is shown in Fig. 2.14.

```
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

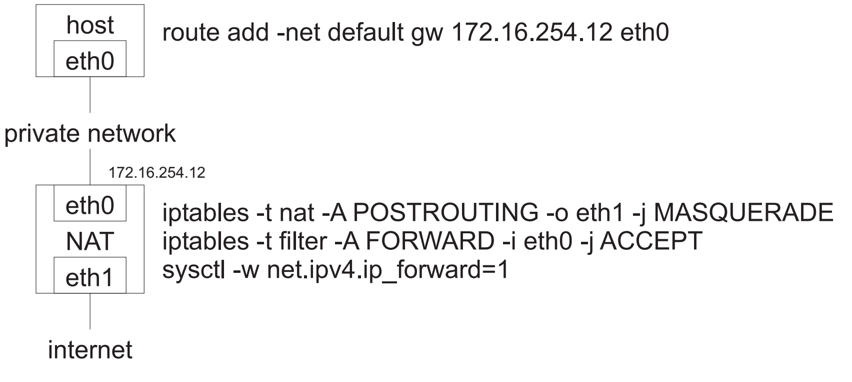

Figure 2.14: NAT configuration

The host providing the NAT service is a gateway to the IP addresses outside the private network range. A packet received at the `eth0` network interface is forwarded to the `eth1` network interface. Thus, it is first tested against the rules in the `FORWARD` chain in the filter table. The following rule ensures acceptance in the `FORWARD` chain (e.g. when the default policy of the `FORWARD` chain is `DROP`):

```
iptables -t filter -A FORWARD -i eth0 -j ACCEPT
```

Finally, the IP forwarding must be enabled. The IP forwarding enables the host providing NAT to act as a gateway forwarding the IP packets from one subnet to another. The kernel parameters are maintained by the `sysctl` command. The IP forwarding status can be obtained by:

```
sysctl net.ipv4.ip_forward
```

If it is disabled (i.e. `net.ipv4.ip_forward=0`), it can be enabled by:

```
sysctl -w net.ipv4.ip_forward=1
```

A change of a kernel parameter made by the `sysctl` command is temporary. The kernel parameters used at the system boot reside in `/etc/sysctl.conf`. The IP forwarding is disabled by default. To permanently enable the IP forwarding, line

```
net.ipv4.ip_forward=1
```

must be added into the `/etc/sysctl.conf` file.

2.14.2 Port forwarding

An IP socket consists of an IP address and a port number (see section 2.4). It represents a communication endpoint for exchanging the data over TCP/IP (see section 2.1). When a TCP/IP packet arrives to the host with a specified IP address, the port number identifies the process to which the packet is delivered. A packet destined to a particular process must first reach the host where the process runs. To avoid this requirement, the port where the process receives the packets can be forwarded to an arbitrary unused port on another host. This technique is called port forwarding. For instance, the port forwarding must be used to enable a private network host behind NAT (see subsection 2.5.1) to provide an Internet service. As shown in Fig. 2.15, the `172.16.254.1:80` port on host A is forwarded to the `212.235.187.71:8000` port on host B. Thus a packet addressed to `212.235.187.71:8000` is received by host B and immediately forwarded to `172.16.254.1:80` on host A. Since host A has no unique IP address, the packets addressed directly to `172.16.254.1:80` can be received only if dispatched inside a private network (e.g. the 172.16.254.0/255.255.255.240 subnet).

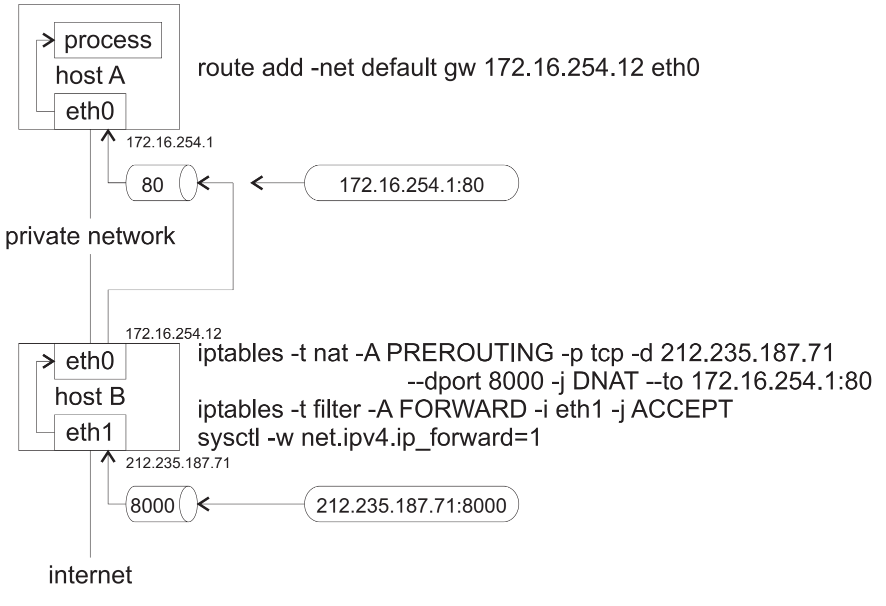

Figure 2.15: Port forwarding

The configuration in Fig. 2.15 can be achieved with a single iptables command on host B. All the TCP transport layer packets received on the `212.235.187.71` IP address port `8000` are forwarded to `172.16.254.1:80`.

```
iptables -t nat -A PREROUTING -p tcp -d 212.235.187.71 --dport 8000
                                        -j DNAT --to 172.16.254.1:80
```

The packet received at the `eth1` network interface on port `8000` is prerouted to `172.16.254.1:80` and thus forwarded to the `eth0` network interface. Therefore, it is also tested against the rules in the `FORWARD` chain in the filter table. The following rule ensures acceptance in the `FORWARD` chain (e.g. when the default policy of the `FORWARD` chain is `DROP`):

```
iptables -t filter -A FORWARD -i eth1 -j ACCEPT
```

Finally, the IP forwarding must be enabled (see page 63). Host B also acts as a gateway to the IP addresses outside the private network range.

The packets received at `212.235.187.71:8000` are merely repacked and sent forward to `172.16.254.1:80`. There is no encryption making the link from host B to host A vulnerable to eavesdropping. Security can be improved by the port forwarding over SSH.

Port forwarding over SSH

The SSH network protocol is used by the `ssh` command for opening a secure command line shell at a remote host (see page 53). It can also be used to encrypt the network traffic belonging to other applications. This is called port forwarding over SSH, or sometimes SSH tunneling, since SSH provides a secured tunnel between two hosts. The SSH port forwarding can be used only for the traffic using TCP on the transport layer. The ports can be forwarded over SSH locally or remotely. The local port forwarding over SSH shown in Fig. 2.16 forwards a port to a local host. A secured tunnel between hosts `A` (e.g. `172.16.254.1`) and `B` (e.g. `172.16.254.2`) is established. The command issued on local host `A`

`ssh -L8000:C:80 user@B`
(or `ssh -L8000:172.16.254.3:80 user@172.16.254.2`)

opens a remote shell and logs into host `B`, as it would without the `-L` option (e.g. `ssh user@B`). The `-L` option additionally forwards the TCP port `80` on host `C` (e.g. `172.16.254.3`) to port `8000` on local host `A`. The forwarded port is active during the SSH session, until logout. When the remote shell is closed, the SSH tunnel ceases to exist.

If the SSH tunnel is opened with the above command, the forwarded port `80` on host `C` can be reached only from local host `A` (e.g. `localhost:8000` or `127.0.0.1:8000`). The `172.16.254.3:80` port is not visible to others as `172.16.254.1:8000`. This restriction can be omitted by the `-g` option permitting any host to connect to a locally forwarded port. The `172.16.254.3:80` port becomes generally visible as the `172.16.254.1:8000` port.

`ssh -g -L8000:C:80 user@B`
(or `ssh -g -L8000:172.16.254.3:80 user@172.16.254.2`)

The connection from host `B`, where the SSH server resides, to host `C` is an ordinary unsecured TCP connection. It is not encrypted and is therefore avoided. Normally, the forwarded port resides on the same host as the SSH server, which is achieved with the following command:

`ssh -L8000:localhost:80 user@B`

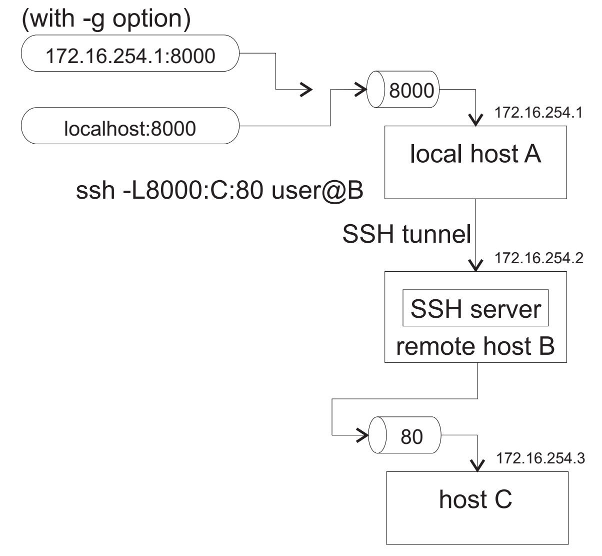

Figure 2.16: Local port forwarding over SSH

(or `ssh -L8000:localhost:80 user@172.16.254.2`)

As depicted in Fig. 2.17, the `localhost:80` port on remote host `B` is forwarded to `localhost:8000` on local host `A`.

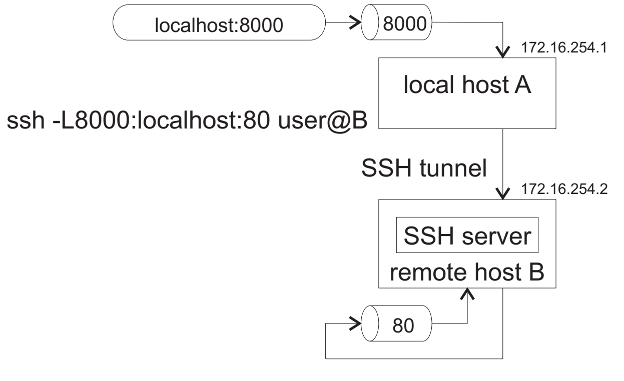

Figure 2.17: Local port forwarding over SSH with no unencrypted connections

The situation is reversed with the remote port forwarding in Fig. 2.18. A command issued on local host `A` forwards a port to remote host `B` where the SSH server resides:

```
ssh -R8000:C:80 user@B
```
(or `ssh -R8000:172.16.254.3:80 user@172.16.254.2`)

The command again opens a remote shell and logs into host `B`. The `-R` option additionally forwards the TCP port `80` on host `C` to port `8000` on remote host `B`. The forwarded port is active during the SSH session, until logout. When the remote shell is closed, the SSH tunnel ceases to exist. The connection from host `A` to host `C` is an ordinary unsecured TCP connection. It is not encrypted and is

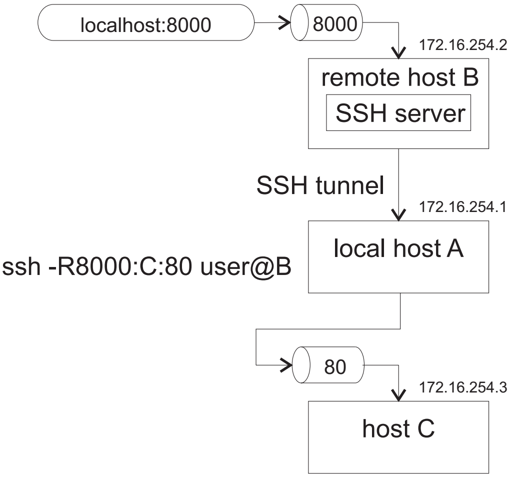

Figure 2.18: Remote port forwarding over SSH

therefore avoided. Normally, the forwarded port resides on local host `A`:

```
ssh -R8000:localhost:80 user@B
```
(or `ssh -R8000:localhost:80 user@172.16.254.2`)

As depicted in Fig. 2.19, the `localhost:80` port on local host `A` is forwarded to `localhost:8000` on remote host `B`.

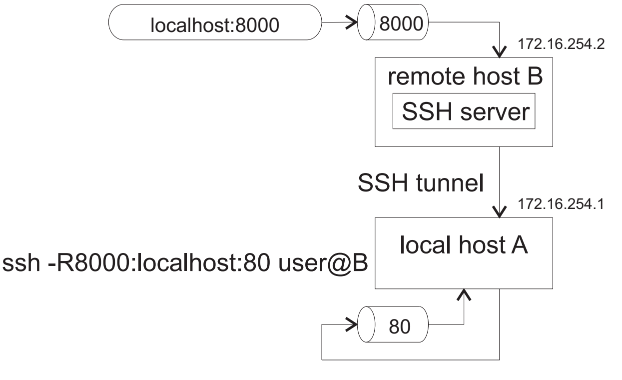

Figure 2.19: Remote port forwarding over SSH with no unencrypted connections

The forwarded port can be reached only from remote host `B` (e.g. `localhost:8000` or `127.0.0.1:8000`). The port is not visible to others as `172.16.254.2:8000`. This restriction can be omitted only by modifying the SSH server configuration on remote host `B`.

Chapter 3

Graphical User Interface (GUI)

The Linux operating systems are completely text-based. This means that GUI is not a part of the operating system. In Linux, GUI is just another program providing graphical features. Such approach enables Linux to run without GUI (e.g. when a host is a server and GUI would just waste its resources). There are many different GUIs available and the user can choose among them. More than one GUI can run on the same machine at the same time (e.g. various look and fill GUIs in different virtual consoles). The X window system usually represents the heart of GUI on the Linux operating systems.

3.1 X window system

GUI in general consists of three parts. The first is an X window system [39]. It is a software enabling the graphical programs to run on Linux. It provides a hardware abstraction layer. Thus, the graphical programs do not need to take care about particular input/output hardware devices attached to the computer (e.g. mouse, keyboard, display). The X window system handles the hardware. The graphical program uses the X window system generalized commands for interaction with the hardware devices.

The X window system provides a place for graphics, but does not control the window with a running graphical program. This is a job for the window manager which is a piece of the software controlling the windows. It is responsible for window moving, hiding, resizing, closing, etc., and what will the mouse actions or keyboard shortcuts cause them. The window manager decides which window is on the top, which one accepts the input, etc.

So far, the X window system provides a place for the graphics and the window manager provides the windows. The additional features, like taskbars, menus, utility programs (e.g. file manager, search tool, text editor, etc.), icons, etc., are delivered by another piece of the software called the desktop manager or desktop environment. Some window managers support the virtual desktops or workspaces.

The X window system is often called X11, since the current major version is 11. It uses a client-server model shown in Fig. 3.1. The X server takes care about the input/output hardware, i.e. the mouse, keyboard and display. On the other side, the X server communicates with the X clients. The X client is a common name for a graphical program. The X server provides the X clients with the user input actions. On the other side, it listens to the X client requests for the graphical output. The window manager and the desktop environment are also the X clients.

The Unix domain sockets (see the table on page 21) are used for communication between the X server and X client when they both run on the same machine [40]. The X server always runs on a local machine, while the X clients can also run on a remote host (Fig. 3.2). The remote X client communicates with the X server

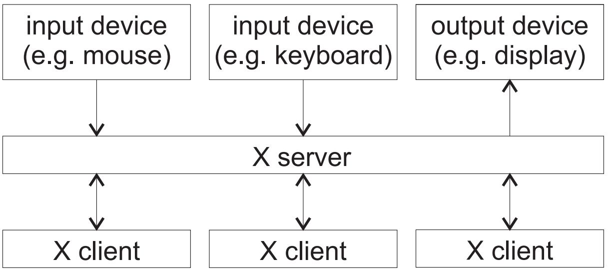

Figure 3.1: Client-server model of the X window system

over TCP/IP (see section 2.1) using the X11 forwarding (see section 3.1.1).

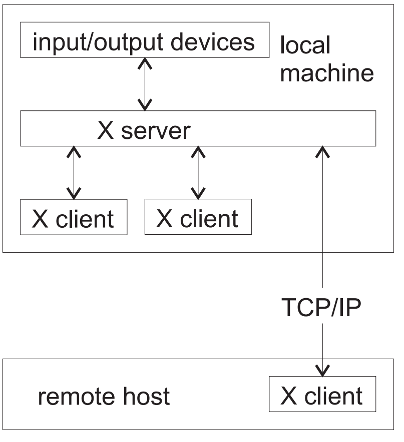

Figure 3.2: X clients running on a local or remote machine

A new single session of the X window system can be started from the text terminal CLI by the `startx` script [11]. It starts a new X server program and, if specified, an X client connected to it. Thus, `startx` can be viewed as a script for starting a single graphical program, i.e. an X client. Arguments immediately following the `startx` command are used to start the X client. The `--` special argument marks the end of the client arguments and start of the server arguments. A particular X server program (e.g. `/usr/bin/X`) and a particular display (see the following paragraph) can be specified. The following examples start a new X session on the `localhost:1.0` display. The first available virtual terminal is used as the `localhost:1.0` display. Usually, that is the eighth virtual terminal. To switch to it, press `Ctrl-Alt-F8`.

`startx -- :1`
: the program named `X` found in the search path (see page 29) is used as an X server by default

`startx xcalc -- /usr/bin/X :1`
: use program `/usr/bin/X` as an X server and the `xcalc` program as an X client

`startx -- :1 -nolisten tcp`
: the X server is started with the `-nolisten tcp` option that prevents the X clients to connect through TCP/IP

The configuration files can be found in the `/etc/X11` directory.

When a new X client is started, it has to know to which X server to connect [40]. Or in other words, it has to know which display (mouse, keyboard) to use. The default display to be used is given in the `DISPLAY` environment variable (the variable has to be exported, see section 1.7) having the following form:

```
host:display.screen
```

The first field, i.e. `host`, is a domain name or an IP address of the computer to which the display (mouse, keyboard) is physically attached. If `host` is not specified, `localhost` is used by default. The second field, i.e. `display` number, represents a set of monitors sharing the same input devices (e.g. the mouse and keyboard). Most computers have only one display though consisting of more than one monitor. Two or more monitors in one display can be configured as a single logical screen allowing the windows to be moved back and forth. Or, the monitors can be configured as individual screens, each with its own windows, which cannot be moved to another monitor. In the latter case, the `screen` number defines which monitor to use. If `screen` is not specified, screen `0` is used by default. `DISPLAY` variable examples:

`:0`	use the `0.0` display on a local machine
`queen.fe.uni-lj.si:1`	use the `1.0` display on `queen.fe.uni-lj.si`
`212.235.187.71:0.1`	use the `0.1` display/screen on `212.235.187.71`

As already mentioned, the Unix domain sockets are used when the X server and X client run on the same machine, i.e. when the domain name or IP address in the `DISPLAY` variable points to the local machine.

Most X client programs accept the `-display` command line option, which temporary overrides the `DISPLAY` variable. Example:

```
freddie@queen:~$ DISPLAY=:0
freddie@queen:~$ export DISPLAY
freddie@queen:~$ xcalc               run xcalc on localhost:0.0
freddie@queen:~$ xcalc -display :1  run xcalc on localhost:1.0
```

An X client started inside the X session connects to a corresponding X server. When an X client is started outside the X session, for instance from another virtual terminal, it cannot connect to the X server by default. The X server has to allow the connection. The connections accepted by the X server can be maintained with the `xhost` command [11]. The `xhost` command has to be executed from an X client (e.g. `xterm`) connected to the X server in question. Example usages of the `xhost` command:

`xhost +local:`
: allow connection for the X clients running on a `local` machine

`xhost +inet:queen.fe.uni-lj.si`
: allow connection for the X clients running on `queen.fe.uni-lj.si`

`xhost`
: print a list of allowed connections

`xhost +`
: turn the access control off, allow connection for all X clients

`xhost -`
: turn the access control on, allow connection only for the X clients running on the listed hosts

`xhost -local:`
: prohibit connection for the X clients running on a `local` machine

`xhost -inet:212.235.187.71`
: prohibit connection for the X clients running on `212.235.187.71`

A typical use of the `xhost` command is shown in Fig. 3.3. First, the X server is started from the text terminal on a local machine. Then, the `xhost` command is executed to allow connections from a remote host. It has to be issued in an X terminal program connected to the X server. The SSH text terminal connection to a remote host follows. The X client program (e.g. `xcalc`) is started on a remote host with a display on a local machine, where its window pops up.

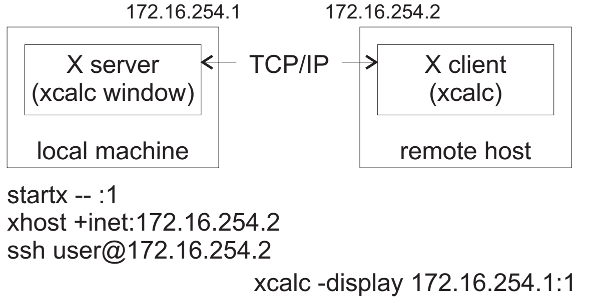

Figure 3.3: Starting an X client on a remote host

A regular TCP/IP X client connection to the X server is not encrypted. So it is vulnerable to eavesdropping and therefore represents a serious security risk. For that reason, the TCP/IP connections of the X clients to the X server are normally closed, which is achieved with the `-nolisten tcp` option. If the X server is started with the `-nolisten tcp` option, then the X clients running on remote hosts cannot connect, despite allowing the connection with the `xhost` command. With TCP/IP closed, the X clients can no longer run on remote hosts. To overcome this awkwardness, the X11 port forwarding over SSH is used. The X11 forwarding over SSH derives from a regular port forwarding over SSH (see subsection 2.14.2).

3.1.1 X11 forwarding over SSH

The X server listens for the X client TCP/IP connections on the port number starting with 6000. The actual port number depends on the display number of the X server (see page 71). If the X server display number is for instance 1 (i.e. the X server display is localhost:1.0), then this X server listens for the TCP/IP connections on port 6001, which is the case in Fig. 3.3. The situation is shown in a more detail in Fig. 3.4.

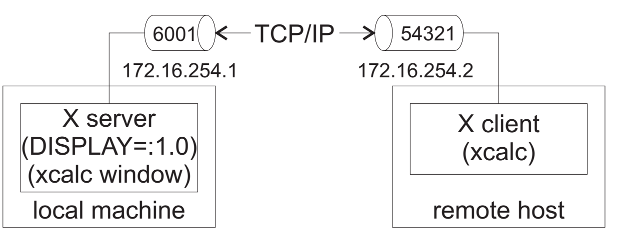

Figure 3.4: X server listening port number

To make the connection between the local and remote host secured, the X server listening port can be forwarded to the remote host over SSH. The X11 port forwarding over an SSH tunnel is shown in Fig. 3.5. It can be achieved by using the `-X` option of the `ssh` command [33].

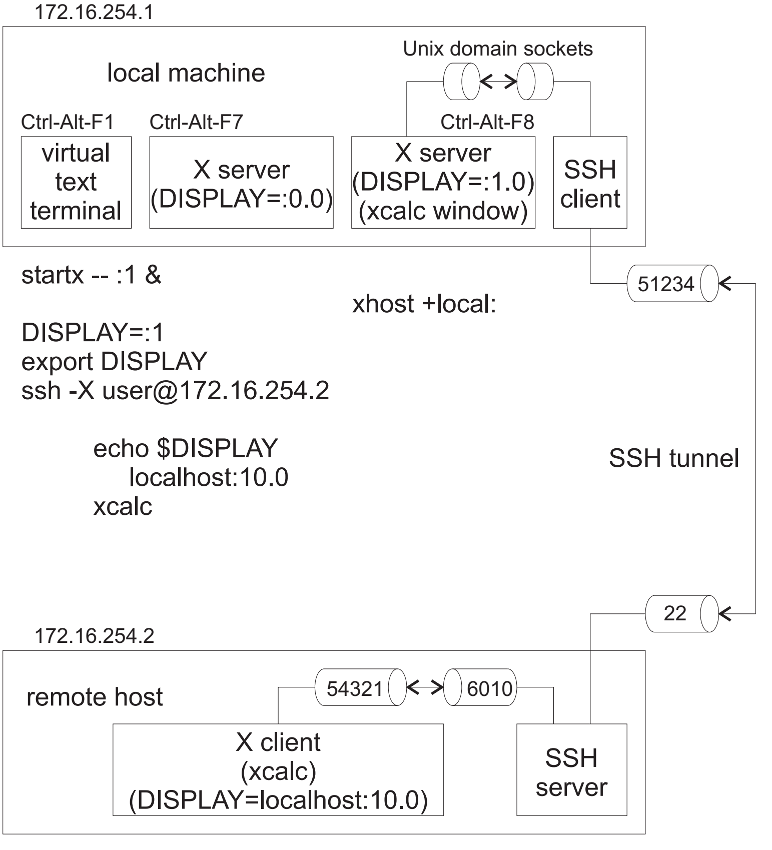

Figure 3.5: X11 port forwarding over a secured SSH tunnel

The `ssh` command opens a remote shell and logs into the remote host (i.e. 172.16.254.2). Additionally, the `DISPLAY` environment variable on the remote host is set (i.e. localhost:10.0) and a corresponding port is forwarded (i.e. the 6010 port). The X client (i.e. `xcalc`) started on the remote host connects to the forwarded port according to its `DISPLAY` value. The connection between the local and remote host is encrypted. The SSH client on a local machine connects to the X server through the Unix domain sockets. An individual X server (there can be more than one) to which the SSH client connects is defined by the SSH client `DISPLAY` value on a local machine. Of course, the X server has to allow the connection.

Chapter 4

Embedded system

A general-purpose computer (e.g. Personal Computer (PC)) is designed to be as flexible as possible. It has to be able to perform a wide range of different tasks, like document editing, performing numerically intensive calculations, e-mail and Internet access, digital media playback, games, acting as a server, etc. On the other side, there are specialized devices designed to perform only one or a few specific functions. The computing unit in such a specialized device is called an embedded system. It is an embedded part of a device. Today, the embedded systems can be found practically everywhere. Our modern life in fact depends on them. They are widely used in telecommunications (e.g. mobile phones, equipment as switches, routers, etc.), consumer electronics (e.g. household appliances, entertainment devices as mp3 players, videogame consoles, digital cameras, etc.), avionics (e.g. inertial guidance systems, Global Positioning System (GPS) receivers, etc.), medical equipment (e.g. vital signs monitoring, medical imaging equipment, etc.), to mention a few.

An embedded system is a specialized computer dedicated to a specific task. Because an embedded system targets a specific task, it can be optimally designed, which reduces its size and cost. As a computing core of an embedded system, various microprocessors, microcontrollers and Digital Signal Processors (DSP) are osed. They always represent a Central Processing Unit (CPU) of an embedded system. A microprocessor is a general name for CPU implemented on a single Integrated Circuit (IC). DSP is a specialized microprocessor adapted to fast processing of sampled digital signals. A microcontroller is a small microprocessor which besides CPU also contains some memory (e.g. usually some flash memory for storing a program and a small amount of the Random Access Memory (RAM) as the program working memory) and/or various input/output peripheral devices (e.g. timer, serial port, etc.) on a single IC. It makes an embedded system even smaller and more compact. A microcontroller with a powerful CPU and substantial amount of memory is also called a System on Chip (SoC). SoC is capable of running a complex software (e.g. the Linux operating system).

A general-purpose computer usually uses a keyboard, mouse and display as a user interface. The embedded systems, on the other hand, often have a very limited user interface or even none. The buttons for issuing the user commands, Light Emitting Diodes (LED) for signaling various states, small Liquid Crystal Displays (LCD) with a simple menu-driven controls, etc., are used. The more sophisticated embedded systems use graphical touch-screen displays, which are approaching to the functionality of a general-purpose user interface, while minimizing the required space. Some embedded systems also provide a remote user interface through a serial (e.g. Universal Asynchronous Receiver/Transmitter (UART), Universal Serial Bus (USB)) or network (e.g. Ethernet) connection. The user interface devices (e.g. the touch-screen display) are no longer needed. The interface of a remote general-purpose computer connected to the embedded system is used instead (e.g. the embedded system running the Linux operating system provides a regular text terminal on the UART serial port; a keyboard and display of a remote general-purpose computer are used as a user interface).

The software is divided into subroutines and processes. A subroutine is a short program performing a specific task. A program taking a considerable time to complete or running indefinitely is referred to as a process. A subroutine is a short process. The software architecture defines when an individual subroutine or process is executed. The following software architecture types or their combination are the ones most commonly used in embedded systems:

Indefinite control loop. An embedded system performs its subroutines one after another. When the last subroutine ends, the first one is called again. An individual subroutine must wait for all the other subroutines to complete to be called again. There are no processes.

Interrupt triggered event handlers. A subroutine, or in this case an event handler, is called at interrupt, which is triggered by an event (e.g. a predefined amount of time has elapsed, data arrived at a serial port, etc.). An individual event handler is called every time its event occurs. In the remaining time when there are no interrupts to be handled, one process can run.

Cooperative multitasking operating system. An operating system enables multiple processes to run seemingly simultaneously by CPU time-sharing. It assigns CPU to a scheduled process. The process occupying CPU is never interrupted by the operating system. It must cooperate and voluntarily return CPU to the operating system after some amount of time. The process resumes its work when next scheduled. One non-cooperative process can hang the whole system by holding CPU for itself. If the processes are simple short subroutines, cooperative multitasking is very similar to an indefinite control loop.

Pre-emptive multitasking operating system. Again, the operating system enables multiple processes to run seemingly simultaneously by CPU time-sharing. These time processes are not aware of time-sharing. The operating system has a full control. It assigns CPU to a scheduled process and interrupts it later to schedule the next process. Thus, the operating system can deal with important external events by immediately assigning the CPU time to the relevant process. It can improve the CPU usage by putting a process (which is for instance waiting for some data) on hold and in the meantime scheduling another process which will fully utilize CPU, etc. A part of the operating system code distributing the CPU time among processes is called a kernel. Since various processes share the same common resources (e.g. memory), the problem of a simultaneous resource access arises. To avoid collisions, some synchronization strategies are required (e.g. message queues, mutexes, semaphores).

Real-time operating system (RTOS). RTOSes belong to a special class of the operating systems. Subroutines and processes have deadlines when they have to complete (e.g. an answer to an event has to be delivered in a specified amount of time). Thus, a scheduling policy of a RTOS primary takes care about when an individual subroutine has to complete. The time is the most important parameter. RTOS which can deterministically guarantee that all subroutines will always complete on time is called a hard RTOS. A soft ROTS, on the other hand, occasionally misses a deadline. The same RTOS can be hard for the high-priority subroutines and soft for the low-priority ones. A RTOS can be cooperative or pre-emptive.

Microkernel and monolithic kernel. A monolithic kernel is a type of the pre-emptive operating system kernel. Every system service like the process and memory management, interrupt handling, input/output communication, file system management, etc., is part of the monolithic kernel. Therefore, the monolithic kernel is in general relatively large (e.g. the Linux kernel). The microkernel is another type of the pre-emptive operating system kernel often used in embedded systems. It reduces the kernel to only the basic process and memory management. Other system services are excluded from the kernel and are provided as normal processes called the servers. When a user process requires a system service, it does not contact the microkernel, but the server process providing the service.

Linux running on an embedded system is called embedded Linux. Linux and embedded Linux are actually the same operating system. Although the embedded Linux kernel is usually tailored to the target embedded system and has therefore a smaller footprint than the complete desktop version of the Linux kernel.

There is no unique recipe of how to build and program an embedded system. A huge variety of developing environments and boards are available for different CPUs and software architectures. Thus, a tutorial/demonstration of the Phytec's phyCORE-i.MX27 development kit [41] running the embedded Linux follows. The CPU is a Freescale i.MX27 microcontroller or SoC based on the ARM926EJ-S architecture (32-bit Reduced Instruction Set Computing (RISC) engine at 400MHz).

4.1 Installing an operating system

In this section, the procedure how to install embedded an Linux operating system on the phyCORE-i.MX27 is described.

Erasing the NOR flash memory and uploading a boot-loader

There is 32MB of the NOR flash memory on the phyCORE-i.MX27 at addresses from `0xc0000000` to `0xc1ffffff`. Erasing and uploading the data to the phyCORE-i.MX27 NOR flash is performed with a special software running on a remote PC connected to the phyCORE-i.MX27 through a UART serial port as shown in Fig. 4.1. The on-board switches must be appropriately set with the power on to boot the phyCORE-i.MX27 over UART[1]. First, the entire 32MB of the NOR flash is erased. This has to be done in several subsequent steps, since only 4MB can be erased at once. The result is a completely bare phyCORE-i.MX27 embedded system without any software. The boot-loader (see subsection 1.1.1) is uploaded into the first 256kB of the NOR flash. A Barebox boot-loader compiled for the i.MX27 microcontroller is used. It can be cross-compiled on a remote PC, or obtained at Phytec [42].

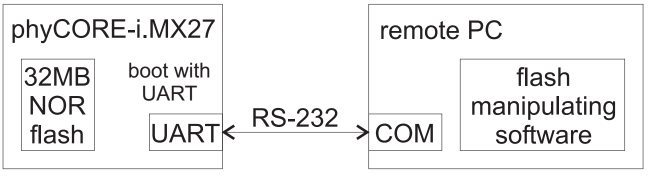

Figure 4.1: Erasing the NOR flash and uploading a boot-loader

[1]To boot phyCORE-i.MX27 with UART, set switches 3, 5 and 7 of S5 into position on.

Barebox boot-loader

A Barebox boot-loader [43] is used in embedded systems to boot the Linux operating system. With the Barebox, the phyCORE-i.MX27 provides a remote text terminal at the UART serial port[2] as shown in Fig. 4.2. Since there is no operating system yet, the phyCORE-i.MX27 cannot actually boot. Since the boot[3] fails, the Barebox shell is started instead.

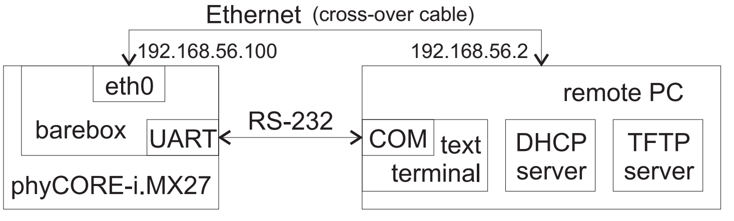

Figure 4.2: phyCORE-i.MX27 embedded system connection to a remote PC

The Barebox shell is a CLI with its own Unix-like commands. Some examples of the Barebox commands are listed below:

`addpart /dev/nor0 256k(barebox)ro,128k(bareboxenv),256k(splash),4M(kernel),-(root)`
: add five partitions to the `/dev/nor0` device

`bootm /dev/nor0.kernel`
: boot the Linux kernel image from `/dev/nor0.kernel`

`dhcp`
: invoke the DHCP client to obtain the IP parameters from the DHCP server

`echo -a /env/config "eth0.ethaddr=$eth0.ethaddr"`
: append the line to `/env/config`

`edit /env/config`
: edit the `/env/config` file, to save changes to RAM, press `Ctrl-D`

`erase /dev/nor0.kernel`
: erase the `/dev/nor0.kernel` partition (to erase the NOR flash partition, it has to be unprotected)

`help`
: print a list of the available commands

`ping 192.168.56.2`
: check the network connection to the `192.169.56.2` host

`protect /dev/nor0.kernel`
: enable the write protection on the `/dev/nor0.kernel` partition (only the NOR flash partitions can be protected)

`readline "enter MAC address:" eth0.ethaddr`
: prompt and read the user input line into the `eth0.ethaddr` variable

`saveenv`
: save the `/env` environment into `/dev/env0` in flash

`tftp linuximage /dev/nor0.kernel`
: get the `linuximage` file from the TFTP server using the TFTP protocol and save it to `/dev/nor0.kernel`

`unprotect /dev/nor0.kernel`
: disable the write protection on the `/dev/nor0.kernel` partition (only the NOR flash partitions can be protected)

---

[2] UART settings: 115200 baud, 8 data bits, 1 stop bit, no parity, no flow control.

[3] To boot phyCORE-i.MX27 from NOR flash, set all switches of S5 into position off.

The Barebox configuration or environment resides in the `/env` directory. It contains the `/env/config` configuration file and scripts in `/env/bin`. The environment is loaded into the RAM disk from `/dev/env0` representing the subsequent 128kB of the NOR flash. In case `/dev/env0` is empty (e.g. at the first start), the Barebox default hard-coded environment is loaded. The modifications made in `/env` are lost after reset. To make them permanent, the environment has to be saved into the subsequent 128kB of the NOR flash (i.e. the `saveenv` command has to be issued).

The `/env/bin/init` initialization script is automatically started at the Barebox start-up. If not interrupted, it starts the `/env/bin/boot` kernel boot script. Both scripts are controlled by the `/env/config` configuration file. The lines in `/env/config` have the following meaning:

Ethernet network interface:

`ip=dhcp`	use the DHCP server to obtain the IP parameters
`eth0.ipaddr`	IP address (not needed if obtained by DHCP)
`eth0.netmask`	subnet mask (not needed if DHCP used)
`eth0.gateway`	default gateway (not needed if DHCP used)
`eth0.serverip`	IP address of the host running servers (e.g. DHCP, TFTP, NFS etc.)
`eth0.ethaddr`	MAC address (printed on the board (e.g. `00:50:c2:5e:00:9a`))

NOR and NAND flash:

`nor_parts`	partition descriptions to be added to `/dev/nor0` (e.g. `/dev/nor0.barebox` (256kB read-only[4]), `/dev/nor0.bareboxenv` (128kB, same as `/dev/env0`), `/dev/nor0.splash` (256kB), `/dev/nor0.kernel` (4MB) and `/dev/nor0.root` (rest, < 28MB))
`nand_parts`	partition descriptions to be added to `/dev/nand0`

Image files:

`bareboximage`	Barebox image file name on the TFTP server
`bareboxenvimage`	Barebox environment image file name on the TFTP server
`splashimage`	splash screen image file name on the TFTP server
`kernelimage`	kernel image file name on the TFTP server
`rootfsimage`	root file system image file name on the TFTP server

Splash screen, Linux kernel and root file system location:

`splash_loc`	splash screen location (possible locations: `nor` ... NOR flash (`/dev/nor0.splash`) and `nand` ... NAND flash (`/dev/nand0.splash.bb`))
`kernel_loc`	Linux kernel location (possible locations: `nor` ... NOR flash (`/dev/nor0.kernel`), `nand` ... NAND flash (`/dev/nand0.kernel.bb`), and `net` ... obtain the kernel from TFTP server)
`rootfs_loc`	root file system location
`rootfs_type`	file system type with the root file system (e.g. `jffs2`)

[4]The five partitions are created by the `addpart` command example on page 78. The read-only property has no effect in the boot-loader. It is an argument to the kernel, thus making the partition read-only in Linux.

`root_mtdblock_nor`	number of the `mtdblock` device with the root file system, used when `rootfs_loc=nor` (e.g. `4` for the `/dev/mtdblock4` device which corresponds to `/dev/nor0.root` in the Barebox)
`root_mtdblock_nand`	number of the `mtdblock` device with the root file system, used when `rootfs_loc=nand` (e.g. `9` for the `/dev/mtdblock9` device which corresponds to `/dev/nand0.root.bb` in the Barebox)
`nfsroot`	root file system directory exported on the NFS server, used when `rootfs_loc=net`

Linux kernel arguments:

`display`	LCD type
`bootargs`	string with arbitrary additional kernel arguments

Uploading the kernel and root file system to the NOR flash

The `/env/bin/boot` boot script tries to boot Linux. Since there is neither the kernel on `/dev/nor0.kernel` nor the root file system on `/dev/nor0.root`, the boot fails. The Linux kernel and root file system for the i.MX27 microcontroller can be cross-compiled on a remote PC, or they can be obtained at Phytec. Uploading to the phyCORE-i.MX27 NOR flash can be performed with a special software from a remote PC (Fig. 4.1) or by using the Barebox shell commands (Fig. 4.2). The first case is slow because of the serial connection. In the second case, TFTP over Ethernet is used. Thus, the remote PC has to be a TFTP server (the TFTP server software can be found in the `tftpd-hpa` package of the Debian Linux distribution, the `tftp-hpa` package for the TFTP client). The TFTP server serves the files in the `/srv/tftp` directory. The Linux kernel and root file system images have to be placed there. If the phyCORE-i.MX27 obtains its IP address by DHCP (i.e. `eth0.ipaddr`, `eth0.netmask` and `eth0.gateway` are not defined), then the remote PC has to be also a DHCP server. With a remote PC properly set (i.e. the TFTP and DHCP servers running, the kernel and root file system files in `/srv/tftp`), the Linux kernel and root file system can be uploaded to the phyCORE-i.MX27 with the following commands:

```
dhcp
unprotect /dev/nor0.kernel
erase /dev/nor0.kernel
tftp linuximage /dev/nor0.kernel
protect /dev/nor0.kernel
unprotect /dev/nor0.root
erase /dev/nor0.root
tftp root.jffs2 /dev/nor0.root
protect /dev/nor0.root
```

The IP parameters are obtained from the DHCP server. The NOR flash kernel and root file system partitions are erased and uploaded with the corresponding images obtained from the TFTP server. The same can be achieved by an update script in the `/env/bin` directory:

```
update -t kernel -d nor
update -t rootfs -d nor
```

The script is controlled by the `/env/config` configuration file.

The same technique can be used for refreshing the Barebox boot-loader or uploading the splash screen:

```
dhcp
unprotect /dev/nor0.barebox
erase /dev/nor0.barebox
tftp barebox-image /dev/nor0.barebox
protect /dev/nor0.barebox
unprotect /dev/nor0.splash
erase /dev/nor0.splash
tftp Splashscreen_i.MX27_240x320.bmp /dev/nor0.splash
protect /dev/nor0.splash
```

Or with the update script:

```
update -t barebox -d nor
update -t splash -d nor
```

The Barebox's environment (i.e. the second 128kB NOR flash partition) can be uploaded in the same way. If not, the default hard-coded environment is used. The environment `/env` directory modifications made from the Barebox shell are lost after reset. To make changes permanent, the environment has to be saved by the `saveenv` command (see page 79). When the Barebox boot-loader is running from the NOR flash, `/dev/env0` points to `/dev/nor0.bareboxenv`.

Uploading to the NOR flash from Linux

At this point, the phyCORE-i.MX27 is installed with an embedded Linux. The NOR flash partitions can be found in the `/dev` directory as Memory Technology Devices (MTD). MTDs come in two flavors, i.e. as character and block devices (see table on page 21). The data from/to a character device (e.g. keyboard) is read/written one character at a time. The character devices do not use buffering and usually do not support a random access. The data from/to a block device (e.g. hard disk) is read/written in blocks. The block devices in general use buffering (i.e. the blocks are accessed through a cache memory) and support a random access. NOR flash MTDs on phyCORE-i.MX27 are:

`/dev/mtd0` - `/dev/mtd4`
: NOR flash partitions as character devices

`/dev/mtd0ro` - `/dev/mtd4ro`
: NOR flash partitions as read-only character devices

`/dev/mtdblock0` - `/dev/mtdblock4`
: NOR flash partitions as block devices

The boot-loader, its environment, splash screen and Linux kernel can be replaced (e.g. with a newer version) from Linux. The embedded Linux `flash_eraseall` command erases an MTD character device. Of course, the device has to have a write permission. Note that the boot-loader is usually marked as read-only (see the footnote on page 79) and therefore cannot be erased by the `flash_eraseall` command. New contents can be copied to the erased device with the `dd` command using block devices. For instance, the Linux kernel (i.e. the fourth NOR flash partition) is replaced with:

```
scp user@192.168.56.2:/home/user/newlinuximage .
flash_eraseall /dev/mtd3
dd if=newlinuximage of=/dev/mtdblock3
```

First, a new Linux kernel image file is uploaded from a remote PC. Clearly, the SSH server has to run on a remote PC and the SSH client on the phyCORE-i.MX27. Then the fourth partition of the NOR flash is erased and reloaded with the obtained kernel image. This technique cannot be used for replacing the root file system since the Linux kernel is running from it.

4.1.1 Mounting an additional memory

Besides 32MB of the NOR flash, the phyCORE-i.MX27 embedded system also provides 512kB of the Static RAM (SRAM) and 64MB of the NAND flash on-board. The external memory devices, like Secure Digital (SD) memory cards or USB keys, can be used as well. The procedures how to mount various types of the additional memory are described here.

`/etc/fstab` file system table

The `/etc/fstab` file is a file system table. It lists the available partitions and specifies their mounting procedure. The mount command reads the `/etc/fstab` file to find out how a specified device should be mounted. Each line in `/etc/fstab` contains the information about one partition organized in six columns: partition device, mount point, file system type, mount, dump and file system check options. For instance, the line:

```
/dev/mtdblock9 /media/nand jffs2 defaults,noauto 0 0
```

specifies that the `/dev/mtdblock9` device will be mounted to `/media/nand`. It contains a file system of the Journaled Flash File System version 2 (JFFS2) type. It will not be mounted automatically (e.g. at the boot or when the `mount -a` command is issued). Otherwise the default option values will be used. The file system will neither be dumped (i.e. backed up) nor checked. With this line in `/etc/fstab`, the `/dev/mtdblock9` device can be mounted with the command:

```
mount /media/nand
```

which is equivalent to:

```
mount -t jffs2 /dev/mtdblock9 /media/nand
```

Journaled Flash File System version 2 (JFFS2)

First of all, the JFFS2 is designed for being used with the flash memory devices [44], although it can be used on other media as well. It uses compression and decompression on the fly. Thus, the partition capacity is virtually enlarged. For instance, checking the summary size of all files and directories in `/` with the command:

```
du -s -h /
```

reports more than 28MB, which is the root file system partition size (see page 79).

Both mount commands in the previous paragraph presume that the `/dev/mtdblock9` device contains the file system of the JFFS2 type. If not, the mount fails with one exception. The mount succeeds if the device has been previously erased (e.g. with the `flash_eraseall /dev/mtd9` command, see page 82). In this case, an empty JFFS2 is automatically built.

A JFFS2 image can be built from an existing directory tree with the `mkfs.jffs2` command. The structure of the memory device for which the JFFS2 image is created has to be given for optimal performance. For instance, the following command builds a JFFS2 image from the `data` directory in the current directory and saves it to the `image.jffs2` file:

```
mkfs.jffs2 -r ./data -o image.jffs2
```

The created image file can be normally used. But to optimize performance for the phyCORE-i.MX27 NOR flash, its erase block size of 128kB (i.e. `0x20000`) has to be given :

```
mkfs.jffs2 -r ./data -o image.jffs2 -e 0x20000
```

For the phyCORE-i.MX27 NAND flash, the erase block size is 16kB (i.e. `0x4000`). Also, no clean markers are required at the beginning of each block:

```
mkfs.jffs2 -r ./data -o image.jffs2 -e 0x4000 -n
```

The image file is loaded to the previously erased device with the `dd` command:

```
flash_eraseall /dev/mtd9
dd if=image.jffs2 of=/dev/mtdblock9
```

SRAM

SRAM MTDs on phyCORE-i.MX27 are:

`/dev/mtd10`	SRAM as a character device,
`/dev/mtd10ro`	SRAM as a read-only character device, and
`/dev/mtdblock10`	SRAM as a block device.

The `/dev/mtdblock10` device has to contain a mountable file system. A Virtual File Allocation Table (VFAT) file system can be used because of its simplicity. Building a file system and mounting SRAM are performed with the following commands:

```
mkfs.vfat /dev/mtdblock10
mount -t vfat /dev/mtdblock10 /media/sram
```

With the following line in `/etc/fstab`:

```
/dev/mtdblock10 /media/sram vfat defaults,noauto 0 0
```

the last mount command can be abbreviated to:

```
mount /media/sram
```

NAND flash

The NAND flash partitions are defined at the Barebox boot-loader start-up. The configuration is given in the `/env/config` file in the `nand_parts` line (see page

79). The NAND flash MTDs on the phyCORE-i.MX27 are:

`/dev/mtd5` - `/dev/mtd9`
: NAND flash partitions as character devices

`/dev/mtd5ro` - `/dev/mtd9ro`
: NAND flash partitions as read-only character devices

`/dev/mtdblock5` - `/dev/mtdblock9`
: NAND flash partitions as block devices

The file system can be built on any NAND flash partition not marked as read-only. For instance, a JFFS2 is automatically built on the second NAND flash partition when mounted to `/media/nand2` with the following commands:

```
flash_eraseall /dev/mtd6
mount -t jffs2 /dev/mtdblock6 /media/nand2
```

With the following line in `/etc/fstab`:

```
/dev/mtdblock6 /media/nand2 jffs2 defaults,noauto 0 0
```

the last mount command can be abbreviated to:

```
mount /media/nand2
```

NOR flash

If the phyCORE-i.MX27 does not boot from the NOR flash (see subsection 4.1.2), then it can be mounted as an additional memory. The NOR flash partitions are defined at the Barebox boot-loader start-up with the `nor_parts` line in `/env/config` (see page 79) and are visible as the MTD devices in Linux (see page 81).

The file system can be built on any NOR flash partition not marked as read-only. For instance, a JFFS2 is automatically built on the second NOR flash partition when mounted to `/media/nor2` with the following commands:

```
flash_eraseall /dev/mtd1
mount -t jffs2 /dev/mtdblock1 /media/nor2
```

With the following line in `/etc/fstab`:

```
/dev/mtdblock1 /media/nor2 jffs2 defaults,noauto 0 0
```

the last mount command can be abbreviated to:

```
mount /media/nor2
```

SD memory card

The Linux operating system reports a new `/dev/mmcblk0` device when an SD memory card is inserted into a slot. A fresh card does not contain any partitions created with the `fdisk` command:

`fdisk /dev/mmcblk0`

`n` add a new partition

`p`	primary partition
`1`	partition number
	the first and last sector define the partition size and its place on the card
`t`	change the partition system identification
	select the partition
`83`	Linux identification
`w`	write the changes to the memory card and exit `fdisk`

The above procedure creates one partition on the `/dev/mmcblk0` memory card. The partition is visible as the `/dev/mmcblk0p1` device. It does not contain a file system yet. The file system of the `ext2` type (second extended file system) is created with the `mkfs.ext2` command. Finally, it can be mounted.

```
mkfs.ext2 /dev/mmcblk0p1
mount -t ext2 /dev/mmcblk0p1 /media/sdcard
```

With the following line in `/etc/fstab`:

```
/dev/mmcblk0p1 /media/sdcard ext2 defaults,noauto 0 0
```

the last mount command can be abbreviated to:

```
mount /media/sdcard
```

USB key

The Linux operating system reports a new `/dev/sda` device when an USB mass storage device is inserted into a slot. A fresh key does not contain any partitions created with the `fdisk` command:

```
fdisk /dev/sda
```

`n`	add a new partition
`p`	primary partition
`1`	partition number
	the first and last sector define the partition size and its place on the key
`t`	change the partition system identification
	select the partition
`83`	Linux identification
`w`	write the changes to the memory card and exit `fdisk`

The above procedure creates one partition on the USB key `/dev/sda`. The partition is visible as a `/dev/sda1` device. It does not contain a file system yet. The file system of the `ext2` type is created with the `mkfs.ext2` command. Finally, it can be mounted.

```
mkfs.ext2 /dev/sda1
mount -t ext2 /dev/sda1 /media/usb
```

With the following line in `/etc/fstab`:

```
/dev/sda1 /media/usb ext2 defaults,noauto 0 0
```

the last mount command can be abbreviated to:

```
mount /media/usb
```

NFS

The files on a remote PC can be accessed through NFS (see section 2.11). Of course, the remote PC has to be an NFS server and the phyCORE-i.MX27 has to be an NFS client. The directory (e.g. `/home/user/dir`) to be accessed from the phyCORE-i.MX27 has to be exported in the `/etc/exports` file on the remote PC. Usually, the `rw`, `no_root_squash` and `sync` options are used. With everything in place, the exported directory can be easily mounted into the phyCORE-i.MX27 directory tree with:

```
mount 192.168.56.2:/home/user/dir /media/nfs
```

NFS mounted on the phyCORE-i.MX27 is especially handy during software development since every change made on a remote PC instantly appears on the embedded system.

4.1.2 Booting from other devices

The boot-loader, operating system kernel and root file system are all uploaded into the NOR flash. Thus, the phyCORE-i.MX27 embedded system boots from the NOR flash. How to place and configure a boot-loader, kernel or root file system to other devices (i.e. NAND flash, SD memory card, USB key and NFS) is described here.

Uploading to the NAND flash

There is 64MB of the NAND flash memory on the phyCORE-i.MX27 at the addresses from `0x00000000` to `0x03ffffff`. Erasing and uploading the data to the phyCORE-i.MX27 NAND flash can be performed from the Barebox shell (see page 78) or from Linux.

First, the entire 64MB of the NAND flash is erased. The NAND flash is configured according to the `nand_parts` line in `/env/config` (see page 79). Thus, the following devices are created in the Barebox `/dev` directory:

`nand0`	whole NAND flash device
`nand0.xxx`	partition named `xxx` (defined by `nand_parts`)
`nand0.xxx.bb`	bad block-aware partition named `xxx`

The whole NAND flash is erased by the Barebox shell command[5]:

```
erase /dev/nand0
```

or it can be erased partition by partition:

[5]Note that the NAND flash cannot be protected.

```
erase /dev/nand0.barebox.bb
erase /dev/nand0.bareboxenv.bb
erase /dev/nand0.splash.bb
erase /dev/nand0.kernel.bb
erase /dev/nand0.root.bb
```

The boot-loader, splash screen, kernel and NAND root file system images are obtained from the remote PC and uploaded to the NAND flash partitions by the `tftp` commands:

```
dhcp
tftp barebox-image /dev/nand0.barebox.bb
tftp Splashscreen_i.MX27_240x320.bmp /dev/nand0.splash.bb
tftp linuximage /dev/nand0.kernel.bb
tftp root.jffs2 /dev/nand0.root.bb
```

If the phyCORE-i.MX27 IP configuration is not static, then its IP address is obtained by DHCP. Clearly, the remote PC has to be a TFTP and DHCP server (see Fig. 4.2). The same can be achieved by the update script in the `/env/bin` directory:

```
update -t barebox -d nand
update -t splash -d nand
update -t kernel -d nand
update -t rootfs -d nand
```

The script is controlled by the `/env/config` configuration file.

The Barebox environment (i.e. the second 128kB NAND flash partition) can be uploaded in the same way. If not, the default hard-coded environment is used. The environment `/env` directory modifications made from the Barebox shell are lost after reset. To make the changes permanent the environment has to be saved by the `saveenv` command (see page 79). When the Barebox boot-loader is running from the NAND flash, `/dev/env0` points to `/dev/nand0.bareboxenv.bb`.

The same can be achieved from Linux. The NAND flash partitions must not be marked as read-only (e.g. the boot-loader partition is usually read-only), otherwise they cannot be erased and uploaded. The images are again obtained from the remote PC, this time by the `scp` commands. To erase the entire 64MB of the NAND flash, all the NAND partitions have to be erased (see page 83). The images are uploaded by the `dd` commands:

```
scp user@192.168.56.2:/home/user/barebox-image .
flash_eraseall /dev/mtd5
dd if=barebox-image of=/dev/mtdblock5
rm barebox-image
flash_eraseall /dev/mtd6
scp user@192.168.56.2:/home/user/Splashscreen_i.MX27_240x320.bmp .
flash_eraseall /dev/mtd7
dd if=Splashscreen_i.MX27_240x320.bmp of=/dev/mtdblock7
rm Splashscreen_i.MX27_240x320.bmp
scp user@192.168.56.2:/home/user/linuximage .
flash_eraseall /dev/mtd8
dd if=linuximage of=/dev/mtdblock8
rm linuximage
```

```
scp user@192.168.56.2:/home/user/root.jffs2 .
flash_eraseall /dev/mtd9
dd if=root.jffs2 of=/dev/mtdblock9
rm root.jffs2
```

This time the remote PC has to be an SSH server, while the phyCORE-i.MX27 is an SSH client.

The Barebox environment can be uploaded in the same way. Note that the root file system NAND partition cannot be erased and uploaded if the Linux kernel is running from it.

Creating a root file system on an external device

Besides the NOR and NAND flash, the root file system can be located on external memory devices like the SD memory card, USB key or NFS. The root file system is created on an external device simply by expanding the `root.tgz file` which can be built on a remote PC or obtained at Phytec. The expansion is performed by the `tar` command, e.g:

```
cd /media/sdcard
tar -xvzf /media/nfs/root.tgz
```

The above two commands presume that the SD memory card is mounted on `/media/sdcard` and that the remote PC directory with `root.tgz` is exported and mounted as NFS on `/media/nfs`. The root file system is expanded to the SD memory card.

Boot configurations

At power on, the phyCORE-i.MX27 loads and runs the boot-loader code from either the NOR or NAND flash. On page 78, the Barebox boot-loader is started from the NOR flash. To run the boot-loader from the NAND[6] flash, the on-board switches have to be appropriately set.

The boot-loader runs the `/env/bin/boot` script which starts the Linux kernel according to the `kernel_loc` line in `/env/config` (see pages 79 and 79). The kernel is placed to either the NOR or NAND flash, or it is uploaded from a remote PC. In the latter case, the remote PC has to be properly set (as on page 80, Fig. 4.2), the kernel image file name is given in the `kernelimage` line of `/env/config`. Examples:

`kernel_loc=nor`	start the kernel from `/dev/nor0.kernel` (the kernel partition has to be listed in `nor_parts`)
`kernel_loc=nand`	start the kernel from `/dev/nand0.kernel.bb` (the kernel partition has to be listed in `nand_parts`)
`kernel_loc=net`	upload the kernel image from the `eth0.serverip` host

Finally, the root file system is mounted according to the `rootfs_loc` line in `/env/config` (see page 80). The `/env/bin/boot` script is prepared for the root file system to be found on either the NOR or NAND flash or NFS. When placed on an external memory device like the SD memory card or USB key, a few lines should be added. Examples:

[6]To boot phyCORE-i.MX27 from the NAND flash, set switch 4 of S5 into position on.

the root file system on the NOR flash:

`rootfs_loc=nor`	
`root_mtdblock_nor=4`	the number of the `mtdblock` device with the root file system according to the `nor_parts` list (e.g. `/dev/mtdblock4`)

the root file system on the NAND flash:

`rootfs_loc=nand`	
`root_mtdblock_nand=9`	the number of the `mtdblock` device with the root file system according to the `nor_parts` and `nand_parts` lists (e.g. `/dev/mtdblock9`)

the root file system on NFS:

`rootfs_loc=net`

`nfsroot=$eth0.serverip:/home/user/phycore/nfsroot`

the exported directory with the root file system

the root file system on the SD memory card:

`rootfs_loc=mmc`

the code in `/env/bin/boot`, the kernel arguments introducing the `ext2` root file system on the `/dev/mmcblk0p1` partition on the SD card added:

```
if [ x$rootfs_loc = xmmc ]; then
  bootargs="$bootargs root=/dev/mmcblk0p1"
  rootfs_type=ext2
else
     if [ x$rootfs_loc = xnand ]; then
          ...
     if [ x$rootfs_type = xubifs ]; then
          ...
fi
```

the root file system on the USB key:

`rootfs_loc=usb`

the code in `/env/bin/boot`, the kernel arguments introducing the `ext2` root file system on the `/dev/sda1` partition on the USB key added:

```
if [ x$rootfs_loc = xusb ]; then
  bootargs="$bootargs root=/dev/sda1"
  rootfs_type=ext2
else
     if [ x$rootfs_loc = xnand ]; then
          ...
     if [ x$rootfs_type = xubifs ]; then
          ...
fi
```

The `/env/bin/boot` script can be started manually from the Barebox shell. It accepts one argument (i.e. `nor`, `nand` or `net`). If the argument is not given, `/env/bin/boot` starts the kernel which mounts the root file system, as defined in `/env/config`. Otherwise, the `kernel_loc` and `rootfs_loc` parameters are altered (i.e. to `nor`, `nand` or `net`, respectively). Additional argument values can be defined to accommodate an arbitrary combination of `kernel_loc` and `rootfs_loc`. For instance, to upload the kernel from the remote PC and use the root file system on the SD memory card when `/env/bin/boot` is called with the `netmmc` argument, add the following code:

```
if [ x$1 = xnetmmc ]; then
  kernel_loc=net
  rootfs_loc=mmc
fi
```

4.1.3 Accessing an embedded system over the network

At this point, an embedded Linux operating system is installed on the phyCORE-i.MX27. If the phyCORE-i.MX27 is connected to the network with a unique IP address obtained over DHCP or statically assigned, it can be accessed from a remote host. The command line shell can be opened over SSH (see page 53) with the command:

```
ssh root@phycore
```

which is equivalent to the remote text terminal at the UART serial port. In the above command, `phycore` is the host name of the phyCORE-i.MX27 embedded system. To be resolved into its IP address, it should be listed in the remote hosts `/etc/hosts` file (see section 2.7), e.g.:

```
192.168.56.100    phycore
```

4.2 Audio and video

The Advanced Linux Sound Architecture (ALSA) is an interface to the sound-related hardware [45]. It is a Linux kernel component. ALSA is a software framework (i.e. a collection of the software libraries) providing device drivers and Application Programming Interface (API) to the sound-related hardware. The command-line `aplay` sound file player can be used for playback simple audio streams with the ALSA sound card driver:

```
aplay audio.wav
```

To capture the audio stream command-line sound file, the `arecord` recorder is available. The following command captures the audio stream on the `hw:0,1` device (e.g. a microphone) for five seconds:

```
arecord -d 5 -D hw:0,1 -c 2 -f S16_LE audio.wav
```

A two-channel stream is written into the `audio.wav` file in the `S16_LE` (signed 16-bit little endian) format.

To configure the ALSA sound settings and adjust the volume, use the `alsamixer` graphical program.

The MP3-encoded audio files can be played with the `madplay` (MAD stands for the MPEG Audio Decoder, since MP3 is an MPEG-2 audio layer III, the standard was developed by the Moving Picture Experts Group (MPEG)) command-line decoder and player:

```
madplay audio.mp3
```

The GStreamer is a framework for creating streaming media applications [46] (e.g. media players). It is based on plug-ins linked and arranged in a pipeline. An element in the pipeline (i.e. plug-in) has a source or a sink, or both. The source generates a stream for the next element in the pipeline and the sink receives the stream from the previous one. The GStreamer pipeline can be created with the command-line `gst-launch` tool. Plug-ins are connected with exclamation marks, e.g.:

`gst-launch audiotestsrc freq=1000 wave=1 ! alsasink`
: `audiotestsrc` plug-in generates a 1kHz square testing signal played by the `alsasink` plug-in (i.e. sound card)

`gst-launch videotestsrc ! videoflip ! ffmpegcolorspace ! fbdevsink`
: `videotestsrc` plug-in generates a test screenshot rotated by the `videoflip` plug-in, converted to an appropriate color space by the `ffmpegcolorspace` plug-in and shown on the `fbdevsink` plug-in

`gst-launch filesrc location=audio.mp3 ! mad ! alsasink`
: the `audio.mp3` (`filesrc`) source is decoded (`mad`) and played (`alsasink`)

`gst-launch filesrc location=audio.mp3 ! mad ! wavenc ! filesink location=audio.wav`
: decode `audio.mp3` and encode it into `audio.wav`

`gst-launch filesrc location=audio.wav ! wavparse ! alsasink`
: decode and play `audio.wav`

Two or more streams of data (e.g. audio and video) are merged together in a container. The container format specifies how various data coexist in one file. An example of a container format is MPEG. The GStreamer is capable of extracting and playing individual streams. After the container has been demultiplexed, multiple streams are created. The command:

```
gst-launch filesrc location=film.mpg ! mpegdemux name=dmx
                                dmx.audio_00 ! queue ! mad ! alsasink
```

is composed of two parts. The `film.mpg` file is demultiplexed into the `dmx` prefixed streams (e.g. `dmx.audio_xx` or `dmx.video_xx`) in the first part. In the second part, the MP3 `dmx.audio_00` stream is decoded and played. Other streams (i.e. `dmx.video_00`) are thrown away. The third part handling the video stream has to be added to play audio and video:

```
gst-launch filesrc location=film.mpg ! mpegdemux name=dmx
                          dmx.video_00 ! queue ! mpeg2dec ! videoflip !
                                          ffmpegcolorspace ! fbdevsink
                                dmx.audio_00 ! queue ! mad ! alsasink
```

The MPEG2 `dmx.video_00` stream is decoded, rotated, converted to an appropriate color space and played on the framebuffer device (e.g. `/dev/fb0` which corresponds to LCD).

4.2.1 Streaming

The audio or video stream or both merged together in a container can be constantly sent over the network to an arbitrary receiver. This is called streaming. The streaming provider transmits the media stream from its media source (e.g. file, microphone, camera, etc.). The streaming receiver presents the received data on the fly.

An audio stream can be for instance sent from the phyCORE-i.MX27 to a remote PC (see Fig. 4.2) and played there. A remote PC is a TCP server listening on port 5000. It decodes and plays the received data:

```
user@remotepc:~$ gst-launch-0.10 tcpserversrc host=192.168.56.2
                                        port=5000 ! mad ! alsasink
```

The remote PC host address is required since the `tcpserversrc` plug-in listens at localhost (i.e. 127.0.0.1, see section 2.8) by default. The phyCORE-i.MX27 is a streaming provider transmitting the audio stream to a remote PC:

```
root@phycore:~# gst-launch filesrc location=audio.mp3 !
                          tcpclientsink host=192.168.56.2 port=5000
```

The streaming direction can be reversed from a remote PC to the phyCORE-i.MX27. Instead of TCP, UDP can be used (see page 42):

```
root@phycore:~# gst-launch udpsrc port=5000 ! mad ! alsasink
user@remotepc:~$ gst-launch-0.10 filesrc location=audio.mp3 !
                              udpsink host=192.168.56.100 port=5000
```

The audio stream is played on the phyCORE-i.MX27. At first, the sound is all right. After a while, it starts racing toward the end, thus skipping parts of the stream. The cause is UDP which does not guarantee the packet deliverance. The packets are sent as fast as possible. Since they arrive ahead of time, most of them are not received. The stream provider has to take care about the transmitting speed when using UDP.

The video stream can be sent also:

```
root@phycore:~# gst-launch tcpserversrc host=192.168.56.100
                                        port=5000 ! mpegdemux name=dmx
                      dmx.video_00 ! queue ! mpeg2dec ! videoflip !
                                        ffmpegcolorspace ! fbdevsink
                              dmx.audio_00 ! queue ! mad ! alsasink
user@remotepc:~$ gst-launch-0.10 filesrc location=film.mpg !
                        tcpclientsink host=192.168.56.100 port=5000
```

Or, for instance, a camera-captured live video[7] is streamed from the phyCORE-i.MX27 to a receiver (e.g. remote PC):

```
user@remotepc:~$ gst-launch-0.10 udpsrc port=5000
              caps=application/x-rtp ! rtpmp4vdepay ! ffdec_mpeg4 !
                          decodebin ! ffmpegcolorspace ! ximagesink
root@phycore:~# gst-launch v4l2src device=/dev/video2 !
    video/x-raw-bayer ! bayer2rgb ! ffmpegcolorspace ! videoscale !
            video/x-raw-yuv ! mfw_vpuencoder codec-type=std_mpeg4 !
  rtpmp4vpay send-config=TRUE ! udpsink host=192.168.56.2 port=5000
```

The video frames are read from a camera device (i.e. `/dev/video2`). The video stream in a raw bayer video format is converted to RGB (Red Green Blue) and

[7]A live camera video is sent to a framebuffer device (i.e. LCD) by the `gst-launch v4l2src device=/dev/video2 ! video/x-raw-bayer ! bayer2rgb ! ffmpegcolorspace ! fbdevsink` command.

further to a YUV color space. Then it is encoded into the MPEG-4 video format payloaded as real-time protocol packets. The packets are sent by UDP. The receiver extracts the MPEG-4 video from the real-time protocol packets and decodes and displays it in a window.

4.3 Developing an embedded application

The embedded Linux operating system is installed on the Phytec phyCORE-i.MX27 development kit. In general, full functionality of the Linux operating system is therefore available (see Chapters 1 and 2). Thus, the phyCORE-i.MX27 can execute the shell scripts, run the C programs, or act as an HTTP and PHP server, etc., to name a few possibilities. The shell scripts can be used without any restrictions (as described in section 1.9). The HTTP and PHP servers are also already installed and running by default. A small memory footprint `lighttpd` and the `php5-cgi` daemons are used as the HTTP and PHP servers. Their configurations can be found in the `/etc/lighttpd` and `/etc/php5` directories. The HTTP, JavaScript or PHP documents can be hosted with no further action.

A few words about compiling and debugging the C and C++ programs for an embedded system can be found in the following paragraphs. If an embedded system is viewed as a Linux box, then the C or C++ source code can be written, compiled and debugged directly on the embedded system as described in section 1.10. To do so, the `gcc`, `g++`, `gdb` and `make` have to be installed, which is normally not the case because of the embedded-system limited resources. Therefore, the executable code is cross-compiled on the remote PC (Fig. 4.2).

Cross-compilation means building an executable code for the target platform (i.e. the phyCORE-i.MX27 with the ARM9 microcontroller) on another platform called the host (i.e. a remote PC with the Intel64 microprocessor). A cross-compiler running on a host platform and producing executable code for the target platform is required. For instance, the `countdown` example from section 1.10 can be compiled for the phyCORE-i.MX27 with the following command on a remote PC[8]:

```
arm-v5te-linux-gnueabi-gcc -o countdown check.c display.c
                                                     countdown.c
```

The `arm-v5te-linux-gnueabi-gcc` program is a cross-compiler. The generated executable file (i.e. `countdown`) is built for the target platform. Therefore, it cannot run on a host platform. It has to be uploaded to the target platform and run there.

Since `gdb` is not available on the target platform, the generated executable file has to be debugged remotely. First, the executable file with the debugging information is required on the host platform:

```
arm-v5te-linux-gnueabi-gcc -g -o countdown_dbg check.c display.c
                                                     countdown.c
```

To enable remote debugging, the `gdbserver` program [18] is launched on the target platform:

```
gdbserver 192.168.56.100:10000 countdown
```

[8] The `arm-v5te-linux-gnueabi-gcc` cross-compiler resides in the `/opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/bin` directory which should be included in the variable `PATH`.

gdbserver is a much smaller program than gdb. It listens on the target platform IP address on port 10000. The debugged program countdown is the final executable file. The debugging information does not need to be included. The remote debugging session starts with the gdb[9] connection from the host platform to gdbserver on the target platform:

```
arm-v5te-linux-gnueabi-gdb countdown_dbg
(gdb) target remote 192.168.56.100:10000
```

The arm-v5te-linux-gnueabi-gdb program is a cross-debugger running on the host platform and used to remotely debug the application running on the target platform. The debugged executable file on the host platform (i.e. countdown_dbg) has to include the debugging information. The embedded application can now be debugged as explained in subsection 1.10.2.

4.4 Programming devices

A few basic primers of the peripheral-device programming can be found in this section. Since the embedded Linux is installed, the devices are in general seen as files in the /dev directory.

4.4.1 System and virtual consoles

The system console is represented as a /dev/console[10] character special file in Linux. It is a text entry and display device for the system administration messages. On the phyCORE.i-MX27 system, the console is connected to the first serial interface (i.e. /dev/ttymxc0). Besides the system console, there are also virtual consoles (see page 5). The following few lines in the C programming language[11] illustrate how the virtual or system console can be used. The console device is opened for reading and writing, read from, written to and closed when done.

```
char buffer[SIZE];
int num_of_bytes, error, descriptor = open("/dev/console", O_RDWR);
     ...
num_of_bytes = read(descriptor, buffer, SIZE);
     ...
num_of_bytes = write(descriptor, buffer, SIZE);
     ...
error = close(descriptor);
```

Of course, the SIZE has to be defined. Declarations of the open(), close(), read() and write() functions are in the fcntl.h header file, which has to be included. The read() and write() functions return the number of bytes actually

[9] The arm-v5te-linux-gnueabi-gdb cross-debugger resides in the /opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/bin directory which should be included in the variable PATH.

[10] In general, the /dev/tty1, /dev/tty2, etc. devices are virtual consoles or virtual terminals (see page 5). The /dev/console device (sometimes linked to /dev/tty0) represents the system console, which is a physical device (e.g. a message written to the system console is displayed there irrespective of the current virtual console). Another magic device is /dev/tty. For an individual process, /dev/tty is its controlling terminal.

[11] For a description of the used functions, see [10, 11].

read or written, respectively. The `close()` function returns zero on success, -1 otherwise.

The console input is typically line buffered[12]. A key-press does not appear until a new-line character. Thus, `read()` waits until `Enter` is pressed and then reads up to the `SIZE` characters. The console attributes (e.g. input processing) can be set by the `ioctl()` function.

```
struct termios console;
     ...
error = ioctl(descriptor, TCGETS, &console);
console.c_lflag &= ~(ECHO | ICANON);
error = ioctl(descriptor, TCSETS, &console);
```

Declaration of the `ioctl()` function is in the `sys/ioctl.h` header file and the `termios` structure is defined in `termios.h`. Both header files have to be included. The `ioctl()` function returns -1 if an error occurs. The `termios` structure contains the terminal information. The current terminal settings are read, the echo[13] and canonical mode are disabled, and the modified settings are saved back. One character at a time is read now, since each typed character arrives immediately.

The console is opened in the blocking mode if not specified otherwise. That means that although the canonical mode is disabled, `read()` waits until the first character arrives. To make the `read()` function return immediately in all cases, the console has to be opened in the non-blocking mode:

```
descriptor = open("/dev/console", O_RDWR | O_NONBLOCK);
```

With the non-blocking mode set, `read()` does not wait for a character (or line in the canonical mode) to arrive. If there is none, it returns -1.

4.4.2 Framebuffer

The first framebuffer is represented as the `/dev/fb0` character special file in Linux. There can be more than one framebuffer devices. A framebuffer provides an abstraction layer to the video display hardware (e.g. LCD). It drives the hardware from the memory buffer containing the current frame of the video data. The application does not need to know the details about the used video hardware. It accesses the hardware through the framebuffer device. In other words, the video driver is not a part of the application itself.

The following few lines in the C programming language[14] illustrate how the framebuffer can be used. Before usage, the framebuffer device has to be opened for reading and writing and mapped into the memory. One red pixel at the location $x = 100$, $y = 200$ is displayed. Afterward, mapping is removed and the framebuffer device closed.

```
int error, descriptor = open("/dev/fb0", O_RDWR);
short int *buffer = mmap(0, 153600, PROT_READ | PROT_WRITE,
                                        MAP_SHARED, descriptor, 0);
     ...
buffer[48100] = 0xf800;
```

[12]The input is processed in a canonical mode, which means that reading is suspended until a delimiter arrives. The delimiters are special characters like End-Of-Line (`EOL`), End-Of-File (`EOF`), etc.

[13]With echo enabled, every typed character is automatically echoed.

[14]For description of the used functions, see [10, 11].

```
    ...
error = munmap(buffer, 153600);
error = close(descriptor);
```

The declarations of the `open()` and `close()` functions are in `fcntl.h` and the declarations of `mmap()` and `munmap()` are in `sys/mmap.h`. Both header files have to be included. The `munmap()` and `close()` functions return zero on success, -1 otherwise.

Mapping of the framebuffer device is `153600` bytes long. The constant derives from the video hardware resolution and the number of bytes per pixel. The phyCORE.i-MX27 has a 240×320 pixel LCD in a highcolor format (16-bit color depth or two bytes per pixel) attached, which explains the number. The information about the framebuffer (e.g. resolution, color depth, etc.) can be retrieved by the `ioctl()` function.

```
struct fb_fix_screeninfo fix;
struct fb_var_screeninfo var;
    ...
error = ioctl(descriptor, FBIOGET_FSCREENINFO, &fix);
error = ioctl(descriptor, FBIOGET_VSCREENINFO, &var);
```

Declaration of the `ioctl()` function is in the `sys/ioctl.h` header file and the `fb_fix_screeninfo` and `fb_var_screeninfo` structures are defined in `linux/fb.h`. Both header files have to be included. The `ioctl()` function returns -1 if an error occurs. The length of the framebuffer memory is in the `fix.smem_len` element (i.e. `153600`). The resolution can be found in `var.xres` (i.e. 240) and `var.yres` (i.e. 320), and the color depth in `var.bits_per_pixel` (i.e. 16).

The `buffer` pointer points to an array of 76800 (i.e. 240×320) 16-bit integers, each describing one pixel. The pixel at the x, y location is described by the `buffer[`y×240+ x`]` element. For the pixel at the location $x = 100$, $y = 200$, the index is `48100`.

A single 16-bit integer in the buffer array defines the color of the corresponding pixel in a highcolor format. The bit masks for the red, green and blue components can be retrieved from the `var.red`, `var.green` and `var.blue` structures obtained by the `ioctl()` function call. The highcolor format is composed from individual color components:

```
highcolor = ((red >> (8 - var.red.length)) << var.red.offset) |
         ((green >> (8 - var.green.length)) << var.green.offset) |
            ((blue >> (8 - var.blue.length)) << var.blue.offset);
```

In the 16-bit highcolor standard, the most significant five bits represent the red component (i.e. `var.read.length` = 5, `var.red.offset` = 11), the next six bits are the green component (i.e. `var.green` = 6, `var.green.offset` = 5), and the last five least significant bits give the blue component (i.e. `var.blue` = 5, `var.blue.offset` = 0). That explains the `0xf800` constant representing bright and pure red.

4.4.3 Touchscreen

On the phyCORE.i-MX27, the touchscreen is accessed through the `/dev/input/event0` special file. It is a part of the input subsystem in the Linux kernel. The input subsystem is an abstraction layer to various input devices, such as the keyboard, mouse, touchscreen, etc. It makes the input events (e.g. keystroke, mouse movement, touchscreen press, etc.) available through a standard file interface.

To avoid writing directly to the touchscreen device, specific software standardized services are provided by the `tslib` library. The library contains a source code with the functions for the touchscreen special file handling (e.g. opening, reading touch events, closing, etc.), plug-in modules performing filtering and smoothing the input data, and testing utilities like calibration, etc. It supports a range of various touchscreen devices.

The following few lines in the C programming language illustrate how the touchscreen can be used. Before usage, the touchscreen device has to be opened and configured. The raw or filtered input data reading is at hand. Afterward, the touchscreen device is closed.

```
struct ts_sample sample[SIZE];
struct tsdev *touchscreen = ts_open("/dev/input/event0", 0);
int num_of_events, error = ts_config(touchscreen);
  ...
num_of_events = ts_read_raw(touchscreen, sample, SIZE);
  ...
num_of_events = ts_read(touchscreen, sample, SIZE);
  ...
error = ts_close(touchscreen);
```

The `SIZE` identifier has to be defined to establish the number of samples in the array. To use the `tslib` data structures and functions, the `tslib.h` header file has to be included[15].
The touchscreen device is opened by the `ts_open()` function for reading only. The configuration is done in the `ts_config()` call which returns zero on success, -1 otherwise. The plug-in modules specified in the configuration file are dynamically linked[16] during the configuration. The default configuration file is `/etc/ts.conf`[17]. Another configuration file can be specified at the runtime by the `TSLIB_CONFFILE` environment variable. The precompiled plug-in library files (i.e. modules) reside in the `/usr/lib/ts` directory[18]. Another plug-in directory can be specified at the runtime by the `TSLIB_PLUGINDIR` environment variable. The `ts_close()` function returns zero on success, -1 otherwise.

The `ts_read_raw()` and `ts_read()` functions read the touchscreen event data. Each event is stored in one `ts_sample` structure holding coordinates, pressure and time of the event (i.e. touch). The `ts_read_raw()` function reads the raw data directly from the touchscreen device not using the linked plug-in modules. On the other hand, the `ts_read()` function returns the values that have passed through a chain of the linked modules performing data filtering and smoothing. If the module linear is linked, then adjustment to the framebuffer (e.g. 240×320 pixel LCD on the phyCORE.i-MX27) coordinates is also done. Calibration of the touchscreen to the framebuffer is carried out by running the `ts_calibrate` utility, which is a part of `tslib`. The utility calculates the calibration coefficients for the touchscreen device specified in the

[15]The source files with the `tslib` functionality (e.g `ts_*.c` files with the used functions) also have to be compiled and linked. They can be specified as the `gcc` input files or the precompiled `tslib` library file can be used.

[16]The `libdl.a` standard library has to be searched by a linker to link the dynamic linking functions. The library can be specified as the `gcc` option (i.e. `-ldl`).

[17]The default configuration filename is specified by the `TS_CONF` identifier, normally defined as the `gcc` option (i.e. `-DTS_CONF=\"/etc/ts.conf\"`).

[18]The default plug-in directory is specified by the `PLUGIN_DIR` identifier, normally defined as the `gcc` option (i.e. `-DPLUGIN_DIR=\"/usr/lib/ts\"`).

`TSLIB_TSDEVICE`[19] environment variable. The coefficients are saved into the `/etc/pointercal` file used by the linear module. Another calibration file can be specified at the runtime by the `TSLIB_CALIBFILE` environment variable.

Both read functions wait until the `SIZE` events happen. To make the `ts_read_raw()` and `ts_read()` functions return immediately, the touchscreen device has to be opened in the non-blocking mode. The second argument of the `ts_open()` function is the non-blocking mode switch:

```
touchscreen = ts_open("/dev/input/event0", 1);
```

The read functions now return immediately, returning the number of events read. If there are no events available, -1 is returned.

4.4.4 Qt for the embedded Linux

Writing the code dealing directly with the framebuffer and touchscreen, as described in subsections 4.4.2 and 4.4.3, can be quite an awkward task. A widget library or GUI toolkit can be used instead. Widget is an element used in graphical applications (i.e. button, label, edit box, etc.). There are many different widget libraries available for the Linux operating system (e.g. Qt, wxWidgets, etc.).

Qt is installed in the root file system for the i.MX27 microcontroller obtained at Phytec or cross-compiled on a remote PC. The Qt widget library is a cross-platform application and user-interface framework [47] (i.e. collection of the software libraries). It uses the standard C++ programming language. A detailed explanation of the C++ programming language and Qt modules far exceeds the scope of this textbook. To demonstrate this technique, a simple example application written in C++ using Qt is presented.

The example in fact consists of two executables: a launcher and an application named Countdown. The launcher serves as a desktop with application icons, although it is an application itself. Its look on the phyCORE.i-MX27 LCD is shown in the left part of Fig. 4.3.

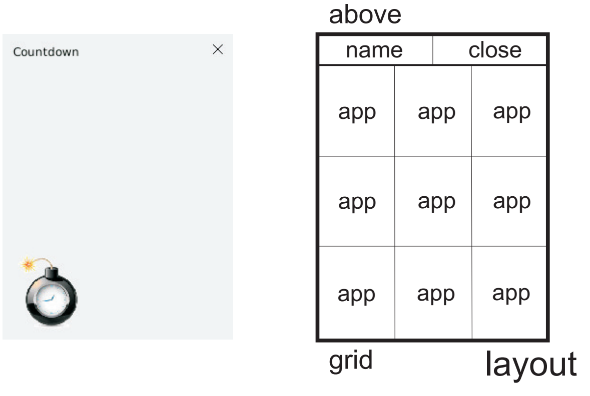

Figure 4.3: Launcher application (left) and its layout structure (right)

There can be up to nine applications on the desktop. Only one place (i.e. bomb icon for the Countdown application) is taken. The application is started by touching the icon. The launcher source code resides in several files.

[19] If the `TSLIB_TSDEVICE` variable is not defined, the touchscreen `/dev/input/event0` is calibrated by default.

```
// main.cpp
#include "launcher.h"

int main(int argc, char *argv[])
{
  QApplication app(argc, argv, QApplication::GuiServer);
  Launcher launcher;
  return app.exec();
}
```

For any GUI application using Qt, there is precisely one `QApplication` object. `QApplication` contains the main event loop. It performs event handling. An event is received from the underlying window system and dispatched to the relevant widget. `QApplication` parses the command-line arguments and sets its internal state accordingly. It must be created before any other object related to the user interface. Event handing is started by `exec()` which enters the main event loop. The `exec()` function returns when the application is closed. For the non-GUI Qt applications, use `QCoreApplication` instead of `QApplication`.

The GUI application requires the window system (i.e. X11, see section 3.1) to provide a hardware abstraction layer. The window system on the other hand uses a significant part of the embedded-system resources. Qt for the embedded Linux eliminates the need for the window system by implementing its own compact window system QWS (QT Window System). QWS is a lightweight user interface server with a small memory footprint. One QWS server is required to which the QWS clients connect. A Qt for the embedded Linux application becomes a QWS server by specifying the `QApplication::GuiServer` type of the `QApplication` object or by running the application with the `-qws` command-line option. Other applications are the QWS clients.

The launcher application main widget is an object of the `Launcher` type. Its constructor creates child widgets and arranges them in a final layout depicted in the right part of Fig. 4.3. The Qt layout objects are used. The child widgets are: the selected application name label, close symbol and application icons. The cursor is set to invisible and the main widget is shown in the full-screen mode without window decorations. The inherited `connect()` function connects the `quit()` signal from the `close` object (i.e. close symbol child widget) to the inherited `close()` slot in this `Launcher` widget. Since they are inherited, the `connect()` function and `close()` slot are not declared in the `Launcher` class. The `close()` slot closes the widget. The signals and slots are used for communication between the objects.

```
// launcher.h
#include <QtGui>

class Launcher : public QWidget
{
  QLabel *name;
public:
  Launcher();
protected:
  void mouseMoveEvent(QMouseEvent *);
};

// launcher.cpp
#include "launcher.h"
```

```
#include "close.h"
#include "app.h"

const char *apps[9][3] = {{"Countdown", ":/bomb.png", "./countdown"}};

Launcher::Launcher()
{
  name = new QLabel();
  Close *close = new Close();

  QHBoxLayout *above = new QHBoxLayout();
  above->addWidget(name, 0, Qt::AlignTop);
  above->addWidget(close, 0, Qt::AlignTop | Qt::AlignRight);

  QGridLayout *grid = new QGridLayout();
  QVBoxLayout *layout = new QVBoxLayout();
  layout->addLayout(above);
  layout->addLayout(grid);

  for(int i = 2; i >= 0; i--) for(int j = 0; j < 3; j++)
  {
    App *app = new App(apps[3 * (2 - i) + j]);
    grid->addWidget(app, i, j);
  }

  QCursor cursor;
  cursor.setShape(Qt::BlankCursor);

  setCursor(cursor);
  setLayout(layout);
  showFullScreen();

  connect(close, SIGNAL(quit()), this, SLOT(close()));
}

void Launcher::mouseMoveEvent(QMouseEvent *event)
{
  QWidget *widget = childAt(event->pos());

  if(widget) name->setText(widget->objectName());
  else name->setText(QString(""));
}
```

The `mouseMoveEvent()` event handler receives the mouse-move events for the widget. The touchscreen stylus (i.e. mouse) position is checked. When a child widget is pointed at, its object name is set as the text of the child name, which thus provides an additional explanation.

A close symbol is represented with the child widget of the `Close` type. The class dealing with signals or slots must inherit `QObject`[20] and must state `Q_OBJECT` in its private section. The `Close` constructor creates a symbol consisting of two crossed lines. When the mouse is released, the `mouseReleaseEvent()` event handler emits the `quit()` signal (which is connected to the launcher `close()` slot).

[20] `Close` inherits `QLabel`, which inherits `QFrame`, which inherits `QWidget`, which inherits `QObject`.

```
// close.h
#include <QtGui>

class Close : public QLabel
{
  Q_OBJECT
public:
  Close();
protected:
  void mouseReleaseEvent(QMouseEvent *);
signals:
  void quit();
};

// close.cpp
#include "close.h"

Close::Close()
{
  QPixmap symbol(10, 10);
  symbol.fill();

  QPainter painter(&symbol);
  painter.drawLine(0, 0, 9, 9);
  painter.drawLine(0, 9, 9, 0);
  painter.end();

  symbol.setMask(symbol.createMaskFromColor(Qt::white));

  setPixmap(symbol);
  setObjectName(QString("Quit"));
}

void Close::mouseReleaseEvent(QMouseEvent *event)
{
  if(rect().contains(event->pos())) emit quit();
}
```

The `App` object represents an application on the desktop. The constructor receives a three-string array with the application name, icon and executable. If the strings are not null, the constructor loads the icon image, sets the object name to the application name and saves the executable. The mouse-released event on this widget starts the application (i.e. `mouseReleaseEvent()` event handler).

```
// app.h
#include <QtGui>

class App : public QLabel
{
  QProcess process;
public:
  App(const char *[]);
protected:
  void mouseReleaseEvent(QMouseEvent *);
```

```
};

// app.cpp
#include "app.h"

App::App(const char *app[])
{
  QPixmap image(80, 80);
  if(app[0])
  {
    image = QPixmap(QString(app[1]));
    image = image.scaled(80, 80);
  } else
  {
    image.fill();
    image.setMask(image.createMaskFromColor(Qt::white));
  }

  setPixmap(image);
  setObjectName(QString(app[0]));

  process.setObjectName(QString(app[2]));
}

void App::mouseReleaseEvent(QMouseEvent *event)
{
  QString name = process.objectName();
  if(rect().contains(event->pos()) && !name.isEmpty())
    process.start(name);
}
```

The icon is obtained from the `:/bomb.png` resource. The colon sign indicates the resource, not the file. The `bomb.qrc` XML-based (eXtensible Markup Language) resource collection file specifies the resources. The resource binary files are stored in the application executable. The `bomb.png` image file is listed as a single resource.

```
<!-- bomb.qrc -->
<!DOCTYPE RCC>
  <RCC version="1.0">
    <qresource>
      <file>bomb.png</file></qresource></RCC>
```

The launcher executable is built from nine input files (i.e. sources, headers and resources): `main.cpp`, `launcher.h`, `launcher.cpp`, `close.h`, `close.cpp`, `app.h`, `app.cpp`, `bomb.qrc` and `bomb.png`. Building the executable consists of:

- generating the C++ source file[21] containing the resource data specified in the `.qrc` file,

[21] C++ file containing the resources is generated with the Qt resource compiler `rcc` (e.g. `rcc -name bomb bomb.qrc -o qrc_bomb.cpp`), which resides in the `sysroot-host/bin` subdirectory of BSP (Board Support Package, can be obtained at Phytec) version (i.e. PD11.1.1) platform (i.e. phyCORE-i.MX27) directory.

- generating the C++ source files[22] containing the meta-object code,
- cross-compiling the C++ source files[23] (the `rcc`- and `moc`-created files included), and
- linking the compiled object files[24] into the final executable.

The above steps can be performed manually. Writing `Makefile` and using the `make` utility is more convenient (see page 34), though, all the more so since `Makefile` can be generated automatically. For that purpose, Qt provides the `qmake` utility. `Makefile` is generated out of the `qmake` project file (`.pro`). The latter contains all the information needed (e.g. list of the input files, building options, etc.) to build the executable. Fortunately, `qmake` can also create a simple project file. The input files found in the current directory are listed by default. Thus, the `launcher` executable is built in three steps: creating the `qmake` project file[25], generating `Makefile`[26] and building the executable with the `make` utility[27]:

```
qmake -project
qmake -spec qws/linux-ptx-g++
make
```

The Countdown application is started by touching the bomb icon (Fig. 4.3). The application displays the time left to the 20th of December 2012 at 24:00 (the left part of Fig. 4.4). Its source code is spread over several files.

```
// main.cpp
#include "close.h"
#include "counter.h"

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
```

[22]The Qt's C++ extensions (e.g. signals and slots) are handled by the Meta-Object Compiler (`moc`). It generates a C++ source file containing the meta-object code for every class with `Q_OBJECT` macro (e.g. `moc close.h -o moc_close.cpp`). The Meta-Object Compiler resides in the `sysroot-host/bin` subdirectory of the BSP version platform directory.

[23]Using the `arm-v5te-linux-gnueabi-g++` C++ cross compiler, which resides in `/opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/bin` directory.

[24]Linker is called by `arm-v5te-linux-gnueabi-g++` C++ cross compiler.

[25]`qmake` utility resides in the `sysroot-cross/bin` subdirectory of the BSP version (i.e. PD11.1.1) platform (i.e. phyCORE-i.MX27) directory. With the `-project` option, it creates a `current_directory_name.pro` project file.

[26]`Makefile` is generated out of the `current_directory_name.pro` project file. `qmake` uses `qt.conf` configuration file, e.g.:

```
# qt.conf
[Paths]
Prefix=.../...BSPversion.../...platform.../sysroot-target/usr
Binaries=.../...BSPversion.../...platform.../sysroot-host/bin
```

`qt.conf` is in the same directory as `qmake`. The Binaries path defines the absolute path to the directory with the required binaries (e.g. `moc`, `rcc`, etc.). The Prefix path is suffixed by the `/mkspecs/` and `-spec` option value. The obtained location specifies the platform configuration directory with the cross-compiler settings for the target. The `QMAKESPEC` variable can be used instead of the `-spec` option, e.g.:

```
export QMAKESPEC=qws/linux-ptx-g++
qmake
```

In case neither the `-spec` option nor `QMAKESPEC` are defined, the default configuration directory (i.e. `prefix/mkspecs/default`) is used.

[27]The C++ cross compiler `arm-v5te-linux-gnueabi-g++` is used. It resides in `/opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/bin` directory, which should be included in the `PATH` variable.

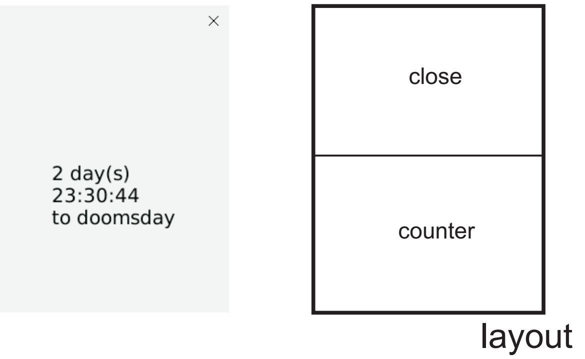

Figure 4.4: Countdown application (left) and its layout structure (right)

```
  Close *close = new Close();
  Counter *counter = new Counter();

  QVBoxLayout *layout = new QVBoxLayout();
  layout->addWidget(close, 0, Qt::AlignTop | Qt::AlignRight);
  layout->addWidget(counter, 0, Qt::AlignTop | Qt::AlignCenter);

  QCursor cursor;
  cursor.setShape(Qt::BlankCursor);

  QWidget countdown;

  QFont font = countdown.font();
  font.setPointSize(20);

  countdown.setFont(font);
  countdown.setCursor(cursor);
  countdown.setLayout(layout);
  countdown.showFullScreen();

  QObject::connect(close, SIGNAL(quit()), &countdown, SLOT(close()));

  return app.exec();
}
```

As in every GUI application using Qt, there is one `QAplication` object handling the event loop. The application main widget (i.e. `countdown`) is a `QWidget` object. The child `close` and `counter` widgets are created and arranged as depicted in the right part of Fig. 4.4. The main widget is shown in the full-screen mode without window decorations, the cursor is invisible and its font size is enlarged. The QObject `connect()` member function connects the `quit()` signal from the `close` object (i.e. close symbol child widget) to the `close()` slot of the application main widget (i.e. `countdown`). The `close()` slot closes the widget.

The `close` child widget is identical to one in the launcher application. The `close.h` and `close.cpp` source files are the same as on page 101.

The amount of the remaining time is refreshed once per second. The counter constructor sets the reference moment (i.e. 20th of December 2012 at 24:00) and starts the timer generating the `timeout()` signal every 1000ms. The signal is connected to the `change()` slot which refreshes the displayed text.

```
// counter.h
#include <QtGui>

class Counter : public QLabel
{
  Q_OBJECT
  QDateTime doomsday;
  QTimer timer;
public:
  Counter();
public slots:
  void change();
};

// counter.cpp
#include "counter.h"

Counter::Counter()
{
  doomsday.setDate(QDate(2012, 12, 21));
  doomsday.setTime(QTime(0 ,0));
  doomsday.setTimeSpec(Qt::UTC);
  connect(&timer, SIGNAL(timeout()), this, SLOT(change()));
  timer.start(1000);
}

void Counter::change()
{
  int secs = QDateTime::currentDateTime().secsTo(doomsday);
  QString text;
  setText(text.sprintf("%d day(s)\n%02d:%02d:%02d\nto doomsday",
   secs / 86400, secs % 86400 / 3600, secs % 3600 / 60, secs % 60));
}
```

The Countdown executable is built in the same way as the launcher (see pages 102 and 103).

To debug the source code, the executable with the debugging information is required. Therefore, the `-g` cross-compiler option has to be used (see section 4.3). The `qmake`-generated `Makefile` specifies the option if the following line is added into the project file:

```
CONFIG += debug
```

Now debugging can be performed as described in section 4.3.

4.4.5 Serial port

There are three serial port connectors (i.e. UART ports) available on the phyCORE-i.MX27 development kit. They can be accessed through the `/dev/ttxmxc0`, `/dev/ttymxc1` and `/dev/ttymxc2` character special files. The first `/dev/ttymxc0` serial port is used as a system console by default (see subsection 4.4.1).

The following few lines in the C programming language[28] illustrate how the serial port can be used. It is opened for reading and writing, configured, read from, written to and closed when done.

```
char buffer[SIZE];
struct termios terminal;
int num_of_bytes, error, descriptor = open("/dev/ttymxc1", O_RDWR);
    ...
memset(&terminal, 0, sizeof(terminal));
terminal.c_cflag = B115200 | CS8;
terminal.c_lflag = ICANON;
terminal.c_iflag = ICRNL;
error = ioctl(descriptor, TCSETS, &terminal);
    ...
num_of_bytes = read(descriptor, buffer, SIZE);
    ...
num_of_bytes = write(descriptor, buffer, SIZE);
    ...
error = close(descriptor);
```

The `string.h` (`memset()`), `termios.h` (`termios`), `fcntl.h` (`open()`, `read()`, `write()` and `close()`) and `sys/ioctl.h` (`ioctl()`) header files have to be included and `SIZE` has to be defined. The serial port configuration is defined by the `termios` structure[29]. The `read()` and `write()` functions return the number of bytes actually read or written, respectively. The `ioctl()` and `close()` functions return -1 if an error occurs.

The carriage-return mapping to a new-line character is not needed in case the canonical mode is not used. Return behavior of the `read()` function is then defined by the `c_cc[VMIN]` and `c_cc[VTIME]`[30] constants. `c_cc[VMIN]` defines the minimum number of characters for the non-canonical read and `c_cc[VTIME]` the timeout in tenths of second. For instance:

- `c_cc[VMIN]` $= 0$ and `c_cc[VTIME]` $= 0$
The `read()` function returns immediately. If no input data is available, then zero is returned. This is a non-blocking read or polling of the serial port. Note that the repeat serial port polling can consume a significant amount of the CPU time.

- `c_cc[VMIN]` $= 0$ and `c_cc[VTIME]` > 0
The `read()` function returns when the required number of characters arrive (i.e. `SIZE` characters), or when the `c_cc[VTIME]` tenths of a second expire. If no input data is available after the `c_cc[VTIME]` expiration, then zero is returned. This is timed read. `c_cc[VTIME]` is the overall timeout.

- `c_cc[VMIN]` > 0 and `c_cc[VTIME]` $= 0$
The `read()` function returns when at least the `c_cc[VMIN]` characters arrive. If

[28] For the description of the used functions, see [10, 11].

[29] The serial port in the example above is configured to the 115200 baud (`B115200` flag), eight data bits (`CS8` flag), one stop bit (default, use `CSTOPB` flag for two stop bits), no parity check (default, use `PARENB` flag to enable parity, even by default, and the `PARODD` flag to use odd parity) and no flow control (default, use `IXON`, `IXOFF` and `CRTSCTS`). The line-buffered input (i.e. canonical mode) is specified (see page 95). The carriage-return (`CR`) character is mapped to a new-line (`NL`), which represents a delimiter in the canonical mode. Otherwise, Enter (i.e. `CR`) does not terminate the input and the `read()` function never returns.

[30] The `c_cc` array holds a list of special control characters and is a member of the `termios` structure.

available, up to the required number of characters (i.e. `SIZE` (> `c_cc[VMIN]`) characters) can be read though. This is a counted read. The `read()` function can block indefinitely while waiting for the `c_cc[VMIN]` characters.

- `c_cc[VMIN]` > 0 and `c_cc[VTIME]` > 0
The `read()` function returns when at least the `c_cc[VMIN]` characters arrive, or when the `c_cc[VTIME]` tenths of a second between two characters expire. The timer is not started until the first character is received. Thus, `read()` can block indefinitely in case the serial line is idle. If available, up to the required number of characters (i.e. `SIZE` (> `c_cc[VMIN]`) characters) can be read. Note that `c_cc[VTIME]` is not overall, but is an inter-character timeout.

If the serial port is opened in the non-blocking mode, then the `read()` function immediately returns in all cases:

```
descriptor = open("/dev/ttymxc1", O_RDWR | O_NONBLOCK);
```

With the non-blocking mode set, `read()` never waits, neither for a new-line character in the canonical mode nor for the minimum number of characters or timeout in the non-canonical mode. If no character is available, -1 is returned, or zero in case of a serial port polling.

4.4.6 Ethernet[31]

The socket interface defines a method for the inter-process communication locally or across the network. The socket is a communication endpoint represented as a regular file descriptor. The traffic is organized in a client-server communication model, where the server waits for the client request and the client requests a service from the server.

The connection-oriented TCP or connectionless UDP transportation protocols above IP are mostly used on the Ethernet (see section 2.1). The client must connect to the server when a connection-oriented protocol is used. Therefore, a socket can be used for communication with only one computer at a time. A connection-oriented TCP provides a reliable and ordered data delivery. On the other hand, a single socket can be used for communication with many different computers when a connectionless protocol is used. But a connectionless UDP does not provide reliability and ordering. The server and client processes using a connection-oriented and connectionless protocol are depicted in Fig. 4.5.

Connection-oriented protocol

The connection-oriented server from Fig. 4.5 is realised in the following lines in the C programming language. The socket is created and bound to the specified `PORT_NUMBER`. The server listens to the socket where up to `NUM_OF_CONN` unaccepted connections can wait. When a connection from the client is accepted, the data can be received and sent with the `read()` and `write()` functions. The connection and socket file descriptors are closed at the end. All the pre-processor macros (`PORT_NUMBER`, `NUM_OF_CONN` and `SIZE`) have to be defined.

```
int socketfd, error, size, connectionfd, num_of_bytes;
struct sockaddr_in server, client;
char buffer[SIZE];
    ...
```

[31] For a description of the functions used in this subsection, see [10, 11].

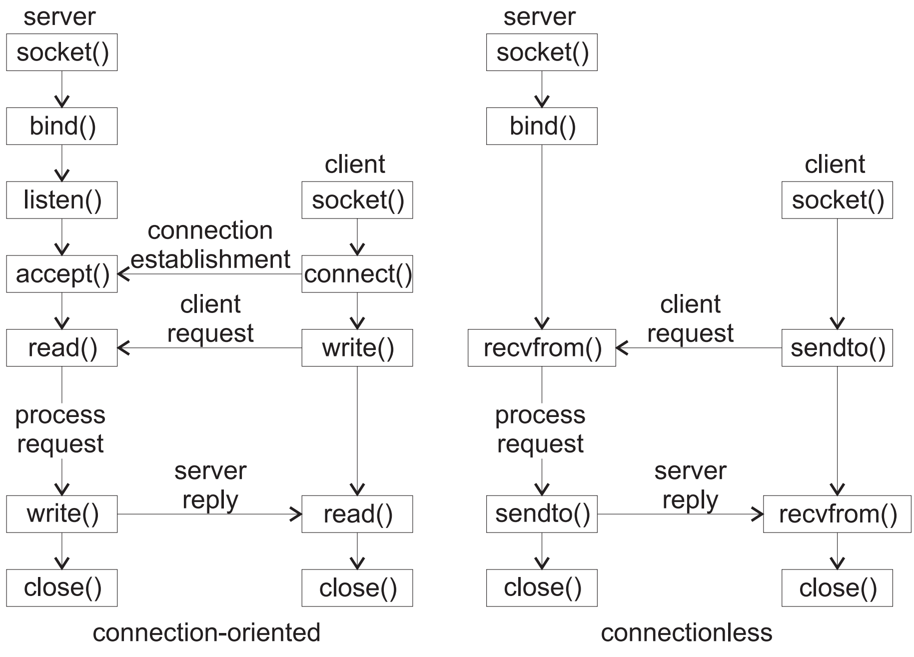

Figure 4.5: Client-server relationship in a connection-oriented (TCP) and connectionless (UDP) protocol

```
socketfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&server, 0, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(PORT_NUMBER);
server.sin_addr.s_addr = htonl(INADDR_ANY);
error = bind(socketfd, (struct sockaddr *)&server,
                                       sizeof(struct sockaddr_in));
    ...
error = listen(socketfd, NUM_OF_CONN);
size = sizeof(struct sockaddr_in);
connectionfd = accept(socketfd, (struct sockaddr *)&client, &size);
    ...
num_of_bytes = read(connectionfd, buffer, SIZE);
    ...
num_of_bytes = write(connectionfd, buffer, SIZE);
    ...
error = close(connectionfd);
error = close(socketfd);
```

The `string.h` (`memset()`) and `netinet/in.h` (`sockaddr_in`, `socket()`, `htons()`, `htonl()`, `bind()`, `listen()`, `accept()`, `read()`, `write()` and `close()`) header files have to be included. The `socket()` function creates a communication endpoint and returns the socket file descriptor. On an error, -1 is returned. The `AF_INET` and `SOCK_ STREAM` arguments specify that IPv4 and connection-oriented TCP are used (see section 2.1). For instance, `AF_UNIX` instead of `AF_INET` would create a Unix domain socket (see page 21 and section 6.7). The `bind()` function sometimes fails at the server rerun since the socket is still hanging in the kernel. The kernel needs a minute or so to clear. The `htons()` and `htonl()` functions convert the short and long integers to the big-endian byte order. Conversion is performed when the host uses the little-endian byte order. IP defines the big-endian as a standard network byte order. The `INADDR_ANY` argument specifies the IP address of the host where the server process runs. If the host has multiple network interfaces (i.e. multiple IP addresses), than the server is allowed to receive the packets destined to any of the interfaces. The `bind()` and `listen()` functions return zero on success, -1 otherwise. The `accept()` function returns the connection file descriptor, -1 on an error. The `shutdown()` function can be used to cut the communication off in only one direction. The `close()` function cuts both ways.

The connection-oriented client from Fig. 4.5 is realized in the following lines in the C programming language. Now, the IPv4 connection-oriented socket is created and the connection to the server with a specified IP address and port number is established. The `read()` and `write()` functions are used to receive and send data. The socket is closed at the end. The `PORT_NUMBER` and `SIZE` macros have to be defined.

```
int descriptor, error, num_of_bytes;
struct sockaddr_in dest;
char buffer[SIZE];
     ...
descriptor = socket(AF_INET, SOCK_STREAM, 0);
memset(&dest, 0, sizeof(struct sockaddr_in));
dest.sin_family = AF_INET;
dest.sin_port = htons(PORT_NUMBER);
dest.sin_addr.s_addr = inet_addr("192.168.56.100");
error = connect(descriptor, (struct sockaddr *)&dest,
                                        sizeof(struct sockaddr_in));
     ...
num_of_bytes = write(descriptor, buffer, SIZE);
     ...
num_of_bytes = read(descriptor, buffer, SIZE);
     ...
error = close(descriptor);
```

The same header files as with the server are required. The `inet_addr()` function converts the IP address string to an integer in the network byte order. The `connect()` function returns zero on success, -1 otherwise.

In the code above, the server process is defined by its host IP address and explicit port number. There are number of network database administration functions (e.g. `gethostbyname()`, `getservbyname()` etc.) that can help to obtain the server process data and much more.

The `accept()` function accepts the next connection in a queue. If there is none, then it waits. Similarly the `read()` function waits until at least one byte is received. As a consequence, the server code on page 107 can handle only one client at a time, which is almost useless. Blocking can be solved in several different ways. One way is the usage of the `fork()` function (see page 131) which creates a new process (i.e. child) duplicating the calling process (i.e. parent). It returns PID of the child process to the parent, -1 on a failure. Zero is returned to the child. For instance, accepting connections can be made continuous regardless of the number of clients being currently served:

```
     ...
while(1)
```

```
{
  pid_t id;
  connectionfd = accept(socketfd, (struct sockaddr *)&client, &size);
  id = fork();
  if(id == 0) break;
  parent process
      ...
  if(...  && children terminated)
  {
    error = close(socketfd);
    terminate parent
  }
}
client handling code (i.e. child process)
error = close(connectionfd);
```

After each accepted connection, a duplication of the server process is created. A new process breaks the `while` loop and starts handling a recently connected client. The original process stays in the `while` loop and waits for the next connection.

Another way is to use the `select()` function to monitor if a file descriptor is ready and the operation can be performed without blocking. The following C code implements monitoring for the `accept()` and `read()` operations. Thus none of the functions waits.

```
    ...
fd_set rfd;
int num_of_fd;
struct timeval interval = {0, 0};
    ...
do
{
  FD_ZERO(&rfd);
  FD_SET(socketfd, &rfd);
  interval.tv_sec = 1;
  num_of_fd = select(socketfd + 1, &rfd, NULL, NULL, &interval);
} while(FD_ISSET(socketfd, &rfd) == 0);
connectionfd = accept(socketfd, (struct sockaddr *)&client, &size);
    ...
do
{
  FD_ZERO(&rfd);
  FD_SET(connectionfd, &rfd);
  interval.tv_sec = 1;
  num_of_fd = select(connectionfd + 1, &rfd, NULL, NULL, &interval);
} while(FD_ISSET(connectionfd, &rfd) == 0);
num_of_bytes = read(connectionfd, buffer, SIZE);
    ...
```

The `select()` function monitors three sets of the file descriptors. The first set is watched for reading (i.e. `rfd`), the second for writing (none given in the code above) and the third for exceptions (none given in the code above). The first argument helps to reduce the number of the file descriptors to be monitored. The file descriptors given in any of the sets must be lower than the first argument. If no file descriptor is ready, then the last argument specifies the amount of the time elapsed before return (i.e. one second). The file descriptor set is handled by the `FD_ZERO()`, `FD_SET()`, `FD_CLR()` and `FD_ISSET()` macros which clear the set, add and remove a descriptor and test if a descriptor is included in the set. The `select()` function modifies the three sets leaving only the ready file descriptors. It returns the total number of the ready descriptors, -1 on an error. In the code above, each `do-while` loop terminates when its file descriptor is ready for a reading operation. Since the socket and connection file descriptors are ready, the `accept()` and `read()` functions do not block, respectively.

In spite of `select()`, reporting a file descriptor as ready for reading, the subsequent read operation may block. This can for instance happen when the data arrives (i.e. `select()` reports a ready status) but is then discarded due to some error (e.g. checksum). Thus, if non-blocking is absolutely required, then a file descriptor should be set into the non-blocking mode, which in fact means that the descriptor can be polled. The `fcntl()` function can be used.

```
    ...
error = fcntl(socketfd, F_SETFL, O_NONBLOCK);
do connectionfd = accept(socketfd, (struct sockaddr *)&client, &size);
while(connectionfd < 0);
    ...
error = fcntl(connectionfd, F_SETFL, O_NONBLOCK);
do num_of_bytes = read(connectionfd, buffer, SIZE);
while(num_of_bytes < 0);
    ...
```

The `fcntl.h` header file has to be included to use `fcntl()`. It returns a nonnegative value on success, -1 on an error. The `O_NONBLOCK` flag is set for the socket and connection file descriptors. Therefore, the `accept()` and `read()` functions return immediately. Polling in the `do-while` loop lasts until the function (`accept()` or `read()`) succeeds.

Connectionless protocol

The following lines in the C programming language represent the connectionless server from a Fig. 4.5. Since the connection establishment is not required, a connectionless server is a simplified version of the connection-oriented one from page 107. The socket is created and bound to the specified `PORT_NUMBER`. The data is received and sent with the `recvfrom()` and `sendto()` functions. The socket file descriptor is closed at the end. The pre-processor `PORT_NUMBER` and `SIZE` macros have to be defined.

```
int socketfd, error, size, num;
struct sockaddr_in server, client;
char buffer[SIZE];
    ...
socketfd = socket(AF_INET, SOCK_DGRAM, 0);
memset(&server, 0, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(PORT_NUMBER);
server.sin_addr.s_addr = htonl(INADDR_ANY);
error = bind(socketfd, (struct sockaddr *)&server,
                                        sizeof(struct sockaddr_in));
    ...
size = sizeof(struct sockaddr_in);
```

```
num = recvfrom(socketfd, buffer, SIZE, 0, (struct sockaddr *)&client,
                                                                &size);
    ...
num = sendto(socketfd, buffer, SIZE, 0, (struct sockaddr *)&client,
                                           sizeof(struct sockaddr_in));
    ...
error = close(socketfd);
```

The `string.h` (`memset()`) and `netinet/in.h` (`sockaddr_in`, `socket()`, `htons()`, `htonl()`, `bind()`, `recvfrom()`, `sendto()` and `close()`) header files have to be included. For a brief function behaviour explanation, see page 108.

The second argument of the `socket()` function is `SOCK_DGRAM`. It specifies the usage of the connectionless UDP. Since the socket is not connected, the data cannot be received by the `read()` function. The `recvfrom()` function has to be used instead. Besides receiving the data, the function also fills the structure with the client IP address and port number. Thus the server knows who to answer. For the same reason, the `sendto()` function has to be used instead of `write()`. The packet destination is given in the structure with the client IP address and port number. The `recvfrom()` and `sendto()` functions return the number of bytes received or sent, respectively, -1 on an error.

The connectionless client from Fig. 4.5 is realized in the following lines in the C programming language. An IPv4 connectionless socket is created. The server IP address and port number have to be prepared before the packet is sent. The `sendto()` and `recvfrom()` functions are used to send and receive the data. The socket is closed at the end. The `PORT_NUMBER` and `SIZE` macros have to be defined.

```
int descriptor, size, error, num;
struct sockaddr_in dest;
char buffer[SIZE];
    ...
descriptor = socket(AF_INET, SOCK_DGRAM, 0);
memset(&dest, 0, sizeof(struct sockaddr_in));
dest.sin_family = AF_INET;
dest.sin_port = htons(PORT_NUMBER);
dest.sin_addr.s_addr = inet_addr("192.168.56.100");
    ...
num = sendto(descriptor, buffer, SIZE, 0, (struct sockaddr*)&dest,
                                           sizeof(struct sockaddr_in));
    ...
size = sizeof(struct sockaddr_in);
num = recvfrom(descriptor, buffer, SIZE, 0, (struct sockaddr *)&dest,
                                                                &size);
    ...
error = close(descriptor);
```

The same header files as with the server are required. The `inet_addr()` function converts the IP address string to the integer in the network byte order. The server process host IP address and port number can be obtained by the network database administration functions (e.g. `gethostbyname()`, `getservbyname()`, etc.).

The `recvfrom()` function waits until at least one byte is received. Therefore, the server can serve one client request at a time. Since the protocol is connectionless, requests from different clients can be served one after another. Thus, more clients can be served quasi simultaneously in spite of the `recvfrom()` blocking. The same blocking solving techniques as with the connection-oriented protocol can be applied, though (see pages from 109 to 111). For instance, using `fork()`, the server can process the client request and wait for another request at the same time. The `select()` function monitors if the socket file descriptor is ready for reading and consequently `recvfrom()` does not block. Or, the socket file descriptor is set into the non-blocking mode by the `fcntl()` function, thus enabling polling.

Chapter 5

Real-time operating system

The major goal of GPOS (General Purpose Operating System) like Linux is an efficient use of the hardware resources and high process throughput. GPOS does not have a deterministic timing behavior, which means that the amount of the time consumed by the operating system is not known in advance. As a consequence, GPOS cannot guarantee that a process with a deadline is executed in time. The deadline can be missed.

RTOS (Real-Time Operating System) does the same thing with the deterministic timing behavior [48]. If the maximum time needed for each of the operating system operations is known in advance, then the operating system can be considered as RTOS. RTOS absolutely guaranteeing these maximum times is called a hard RTOS. A hard RTOS can ensure that all the deadlines are always met. In a hard RTOS, the latencies are deterministic. On the other hand, RTOS guaranteeing these maximum times for most of the time (but not all of the time) is called a soft RTOS. The deadlines can be occasionally missed in a soft RTOS. In a soft RTOS, the latencies are deterministic most of the time, but not all of the time.

From the process point of view, the same RTOS can be either hard or soft, depending on the process priority. In an extreme case, a process with the highest priority can use 100% of the CPU time. Consequently, RTOS does not assign CPU to any other low-priority process. Low-priority processes therefore miss their deadlines. They have to wait until the highest-priority process finishes.

GPOSes are not so strict regarding the priorities. GPOS typically ensures some amount of the CPU time to all the processes. The high-priority processes receive more and the low-priority processes less of the CPU time. But the high-priority processes do not completely block the low-priority ones.

The RTOS in the above paragraph where a high-priority process takes precedence over a low-priority one is pre-emptive. The operating system interrupts the low-priority process in order to assign CPU to the high-priority one. The low-priority process continues when the high-priority process finishes. The opposite is a cooperative model. The operating system cannot interrupt the process and assign CPU to another process in a cooperative model. Once CPU is given to a process, the process has to explicitly return the control to the operating system. The processes must cooperate in the cooperative RTOS. Linux is a pre-emptive GPOS.

5.1 Real-time pre-emptive kernel

As already mentioned, Linux is a pre-emptive GPOS. To make Linux RTOS, its kernel has to be changed. The kernel must be configured as fully pre-emptable. The today's standard Linux kernel (version 2.6) is not fully pre-emptable by default.

For example, when a low-priority process makes a system call, it cannot be pre-empted by a high-priority process. A high-priority process must wait until the system call completes. The situation can be solved by the `CONFIG_PREEMPT` kernel configuration option[1] enabling a pre-emption during the system calls. The `CONFIG_PREEMPT` option reduces the latencies at the cost of a smaller throughput because of more frequent context switching[2].

But even with `CONFIG_PREEMPT`, the kernel is still not fully pre-emptable (e.g. spinlock[3] and RCU[4] (Read-Copy Update) read-side critical sections, interrupt handlers, etc.). An additional pre-emption can be introduced into the Linux kernel by the `CONFIG_PREEMPT_RT` kernel patch[5] (see subsection 1.10.1) [49]. The patch reimplements the kernel locking primitives to be pre-emptable, the priority inversion prevention protocols (see pages 151 and 152), converts the interrupt handlers into the pre-emptable kernel threads, etc. Of course, an appropriate `CONFIG_PREEMPT_RT` patch version has to be applied regarding the kernel version. The patch is not available for all kernel versions. The information about the operating system and the kernel version can be checked by the `uname` (unix name) command:

```
uname -a
```

The `CONFIG_PREEMPT_RT` patched kernel for instance responds with (kernel name, host machine name, kernel version with the compilation date and time, CPU architecture, operating system name):

```
Linux phyCORE 2.6.33.3-rt19 #1 PREEMPT RT Thu Dec 20 23:59:59 UTC
                                            2012 armv5tejl GNU/Linux
```

5.2 Programming a real-time application[7]

An application has to be written in a special way to achieve a real-time behavior in a pre-emptive Linux kernel environment. The deadline must not be missed because of some lengthy system operation. Thus, an application has to make sure that:

- its priority and scheduling policy are appropriately set and
- memory page faults never happen.

[1]The kernel options set before the kernel compilation define the kernel configuration. The kernel shipped with Linux distributions like Debian is compiled with a default set of options. To change an option, the kernel has to be recompiled.

[2]The context switch is a procedure switching CPU from one process to another. It saves the state (e.g. CPU register values, etc.) of the pre-empted process and restores the state of the pre-empting process.

[3]Spinlock is a lock (i.e. infinitive loop) when a process or a thread[6] waits for the locked resource to become available (i.e. unlocked). The process or thread remains active while waiting, doing no progress except for repeatedly checking the resource. This is also called busy waiting.

[4]RCU allows the concurrent reads and updates by maintaining multiple versions of an object. It replaces conventional object locking by a reader or updater, or read-write lock when concurrent reads are allowed (i.e. the reader does not lock the object for other readers).

[5]Patch is an incremental software upgrade containing the differences from the previous version.

[6]Thread is a (sub)process created by another process (see page 129). The difference between the two is that the threads created by the same process share the same address space. Different processes do not.

[7]For the description of the functions used in this section, see [10, 11].

5.2.1 Setting the application priority and scheduling policy

The application priority and scheduling policy are set by the `sched_setscheduler()` function. The policy can be retrieved by the `sched_getscheduler()` and the priority by the `sched_getparam()` function. The application has to run with the super user privileges to make the `sched_setscheduler()` function call work. The priority and scheduling policy of a process can also be found in the 40$^{\text{th}}$ and 41$^{\text{st}}$ field (i.e. `rt_priority` and `policy` fields) of the `/proc/[PID]/stat` file. `[PID]` stands for the PID number of the process. An example code:

```
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>

void terminate(char* msg)
{
  perror(msg);
  exit(0);
}

int main(int argc, char* argv[])
{
  int policy;
  struct sched_param param;
  param.sched_priority = 50;

  /* set process priority and scheduling policy */
  if(sched_setscheduler(0, SCHED_FIFO, &param) == -1)
    terminate("sched_setscheduler() failed");

  /* get scheduling policy */
  policy = sched_getscheduler(0);
  if(policy == -1) terminate("sched_getscheduler() failed");
  printf("policy: %s", policy == SCHED_OTHER ? "SCHED_OTHER" :
                             policy == SCHED_FIFO ? "SCHED_FIFO" :
                      policy == SCHED_RR ? "SCHED_RR" : "unknown");

  /* get priority */
  if(sched_getparam(0, &param) == -1)
    terminate("sched_getparam() failed");
  printf("priority: %d\n", param.sched_priority);   return 0;
}
```

The first argument (i.e. zero) in the `sched_*()` functions denotes the calling process. `SCHED_OTHER` specifies the standard round-robin time-sharing[8] scheduling policy. The process priority is not used in scheduling decisions and is always zero. `SCHED_FIFO` and `SCHED_RR` specify first in first out and round-robin real-time scheduling policies. The process priority can be set from 1 to 99 for both policies. In the `SCHED_FIFO` policy, a high-priority process pre-empts a low-priority one. The processes with the same priority are executed one after another. To start the next process, the previous has to complete. `SCHED_RR` is a slight enhancement

[8]Equal CPU time slices in a circular order are assigned to the processes. No priority is assigned.

of SCHED_FIFO. The processes with the same priority run simultaneously in the round-robin time-sharing scheduling policy.

5.2.2 Process memory

Address space. A virtual memory assigned to a process is called the address space. It is in general divided into the code, data and stack segment. By default, the process address space is not limited, although the limit can be set by setrlimit() and retrieved by the getrlimit() function. The address space limit can also be found in the /proc/[PID]/limits file, where [PID] stands for the PID number of the process. If limited, the address space can be exceeded by the dynamic memory allocation or stack expansion (see page 121). An example code:

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

void terminate(char* msg)
{
  perror(msg);
  exit(0);
}

int main(int argc, char* argv[])
{
  struct rlimit limit;

  /* set address space limit to 1MB */
  limit.rlim_cur = 1024 * 1024;
  limit.rlim_max = RLIM_INFINITY;
  if(setrlimit(RLIMIT_AS, &limit) == -1)
    terminate("setrlimit() failed");

  /* get address space limit */
  if(getrlimit(RLIMIT_AS, &limit) == -1)
    terminate("getrlimit() failed");
  printf("maximum address space size: ");
  if(limit.rlim_cur == RLIM_INFINITY) printf("unlimited\n");
  else printf("%d bytes\n", limit.rlim_cur);
}
```

A code segment, also called text section, holds the executable instructions (i.e. machine code) of the process. The global constant data resides there as well. Its boundaries can be found in 26^{th} and 27^{th} field (i.e. the startcode and endcode fields) of the /proc/[PID]/stat file. The code segment is read-only.

A data segment consists of the data, BSS (Block Started by Symbol) and heap section. The initialized global and static variables reside in the data section. The uninitialized global and static variables are in the BSS section which is set to zero. The uninitialized global and static variables are thus in fact initialized to zero at the process start. A heap section is used for the dynamic memory allocation. Therefore, the heap size varies during the runtime. By default, the maximum data segment size is not limited, although the limit can be set by setrlimit() and retrieved by the getrlimit() function (in the code on page 118 replace

RLIMIT_AS with RLIMIT_DATA). The data segment limit can also be found in the /proc/[PID]/limits file, where [PID] stands for the PID number of the process. The memory is dynamically allocated by the malloc()[9] and released by the free() function. Normally, the memory is allocated in the heap section of the data segment. The current data segment size is defined by the program break adjusted according to the current heap size (Fig. 5.1). The program break is an address where the data segment ends. Thus, it defines the top of the heap. The program break can be increased or decreased by the sbrk() function. It cannot be set beyond the maximum data segment size.

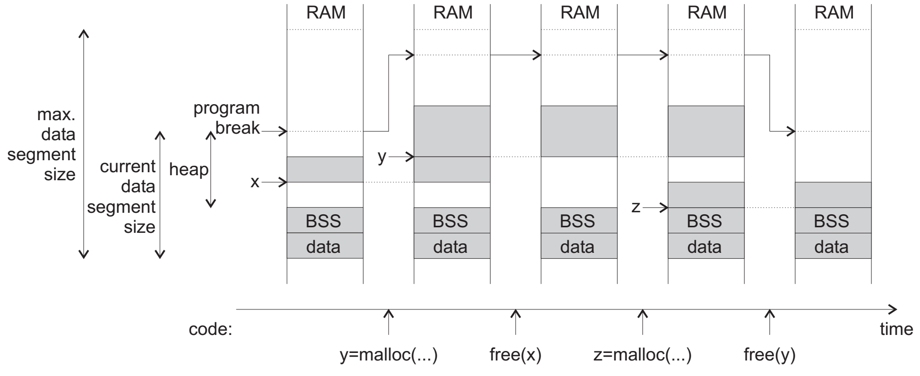

Figure 5.1: Heap section variations in a data segment

Heap management (e.g. increasing and decreasing the program break, fragmentation, etc.) is a part of the memory allocation function (i.e. malloc()) implementation. The kernel in fact does not know about the heap and dynamic memory allocation. It just receives the sbrk() requests sent by the memory allocation function to increase or decrease the process data segment (i.e. program break).
The program break cannot be decreased until the contiguous memory on top of the heap is released. An unreleased block on top of the heap can hold the program break high, although the heap below is unused. To avoid the described situation, the memory can be allocated outside the heap as an individual memory mapping by the mmap() function (see page 95). The mapped memory is in the process address space but is not part of the data segment. It is immediately returned to the kernel when released. The memory allocation function decides when mmap() will be used. The behaviour can be controlled by the mallopt() function. An example code:

```
#include <malloc.h>
#include <stdlib.h>
#include <unistd.h>

void terminate(char* msg, int pf)
{
  if(pf == 0) perror(msg);
  else printf(msg);
  exit(0);
}

void *allocate(int size)
```

[9]Or malloc() similar functions (e.g. realloc()).

```
{
  void *ptr = malloc(size);
  if(ptr == NULL) terminate("malloc() failed", 0);
  return ptr;
}

int main(int argc, char* argv[])
{
  /* get initial program break, heap size is zero */
  void *ptr[3], *brk_start = sbrk(0);

  /* use mmap() instead of sbrk() for requests equal to or */
  /* larger than 50kB if not enough space on heap          */
  int err = mallopt(M_MMAP_THRESHOLD, 50 * 1024);
  /* program break is page-aligned (e.g. page size: 4kB)       */
  /* leave at least 50kB pad of free memory on top of the heap */
  /* at sbrk() call                                            */
  err = err + mallopt(M_TOP_PAD, 50 * 1024);
  /* do not decrease program break until at least 70kB          */
  /* (pad included) of memory on top of the heap is free        */
  /* note: the above is required but not sufficient to decrease */
  /*       program break                                        */
  /*       when sbrk() is actually called depends on free()     */
  /*       implementation                                       */
  err = err + mallopt(M_TRIM_THRESHOLD, 70 * 1024);
  if(err < 3) terminate("mallopt() failed\n", 1);

  /* allocate 1kB using sbrk(), heap size increased to 52kB */
  ptr[0] = allocate(1024);
  printf("heap size: %dkB\n",(sbrk(0) - brk_start) / 1024);

  /* allocate 50kB, enough space on heap, mmap() is not used */
  ptr[1] = allocate(50 * 1024);
  printf("heap start: 0x%08x end: 0x%08x allocated at: 0x%08x\n",
                                        brk_start, sbrk(0), ptr[1]);

  /* allocate 50kB, not enough space on heap, mmap() is used */
  ptr[2] = allocate(50 * 1024);
  printf("heap start: 0x%08x end: 0x%08x allocated at: 0x%08x\n",
                                        brk_start, sbrk(0), ptr[2]);
  free(ptr[2]);

  /* allocate 2kB using sbrk(), heap size increased to 104kB */
  ptr[2] = allocate(2 * 1024);
  printf("heap size: %dkB\n", (sbrk(0) - brk_start) / 1024);

  /* release 52kB on top of the heap, heap size decreased to 52kB */
  free(ptr[2]);
  free(ptr[1]);
  printf("heap size: %dkB\n", (sbrk(0) - brk_start) / 1024);

  free(ptr[0]);

  return 0;
```

```
}
```

The memory allocation fails if there is not enough unused address space available. The program break increase or individual mapping cannot be performed. The memory allocation error occurs[10].

Stack segment. The memory space for the process current data is called stack. The local variables[11], function return addresses, etc.[12] are stored there. The stack is organized as an LIFO (last in first out) data structure (Fig. 5.2). The local variables, return address, etc., are placed on top of the stack at the function call and released at return. Thus the stack size varies during the runtime. A stack and heap usually grow in the opposite directions. The growth direction is platform-dependent. In Fig. 5.2, the stack grows downwards from the higher to the lower addresses. A stack overflow occurs if the maximum stack size is exceeded, which results in a segmentation-violation fault signal sent to the process (see page 26). Segmentation violation also happens if there is no address space left for the stack expansion.

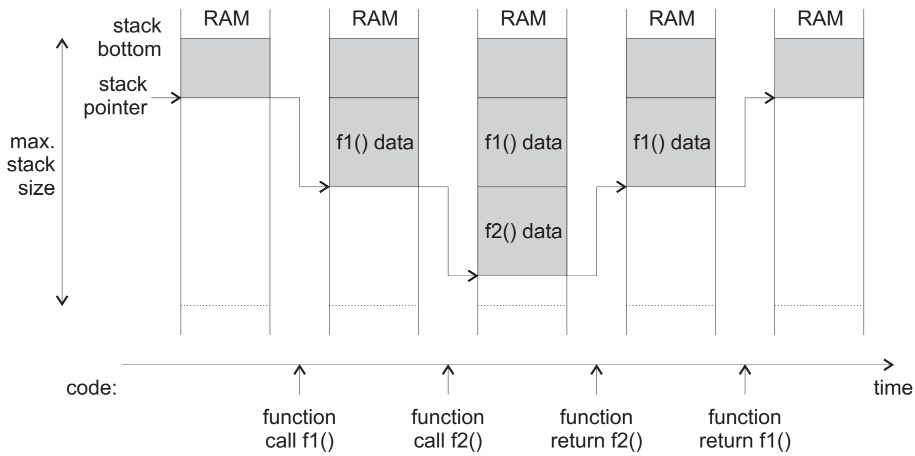

Figure 5.2: Stack

The process stack bottom address can be found in 28th field (i.e. `startstack` field) of the `/proc/[PID]/stat` file, where `[PID]` stands for the PID number of the process. The current top of the stack (i.e. stack pointer value) is in 29th field (i.e. `kstkesp` field) of the same file. By default, the maximum stack segment size is set to 8MB, although the limit can be set by the `setrlimit()` and retrieved by the `getrlimit()` function (in the code on page 118 replace `RLIMIT_AS` with `RLIMIT_STACK`). The maximum stack size can be found in the `/proc/[PID]/limits`.

5.2.3 Preventing the memory page faults

The virtual memory assigned to a process can be larger than the physical memory (i.e. RAM) on the machine. Consequently, the entire address space of the process

[10]The last error number variable errno is set to `ENOMEM` (i.e. not enough space).

[11]The command line arguments are actually local variables of the `main()` function and can be found at the bottom of the stack.

[12]e.g. the local constant data and register values to be restored at the function return also reside on the stack.

cannot be loaded into the physical memory. Only a currently in use portion of the address space is loaded into the physical memory. The rest is stored in some auxiliary memory storage, e.g. hard disk drive. A process is not aware of which parts of its address space are loaded into the physical memory and which are stored elsewhere. The operating system kernel provides retrieving of the requested memory into the physical memory and storing the currently not required memory back to the auxiliary storage. A special hardware MMU (Memory Management Unit) built into CPU is used.

The memory is retrieved and stored in fixed-size blocks called pages[13]. A page fault takes place when a process accesses the memory on a page not loaded into the physical memory. The page has to be retrieved from the auxiliary memory first, which is time expensive. The process has to wait. Therefore, a real-time process can miss its deadline.

Page faults can be avoided by locking all the required memory pages into the physical memory. A locked memory page cannot be paged out into the auxiliary memory. Thus, the pages used by a real-time application have to be locked.

The Linux kernel does not limit the maximum amount of the locked memory for a process with the super-user privileges. The physical memory size is of course an ultimate limit. For an unprivileged process, the limit is set to 64kB and can be lowered by the `setrlimit()` and retrieved by the `getrlimit()` function (in the code on page 118 replace `RLIMIT_AS` with `RLIMIT_MEMLOCK`). The maximum amount of the locked memory can be found in the `/proc/[PID]/limits`, where `[PID]` stands for the PID number of the process. The amount of the currently locked memory can be found in the `VmLck` field of the `/proc/[PID]/status` file.

The entire memory used by a process can be locked with the `mlockall()` function. The function locks the pages with the code, data and stack segment, as well as individual mappings and shared memory. Since the amount of the memory used by a process is normally greater than 64kB, the `mlockall()` function has to be called with the super-user privileges. The memory is automatically unlocked on process termination.

Heap page faults. The page faults because of the dynamically-allocated memory access can be avoided in two ways. In the first approach, all required memory is reserved and locked at the beginning of the code. The memory is never released. No page fault can happen since all the allocated memory is locked. The memory requirements have to be known in advance. Thus, the dynamic memory allocation is not really dynamic. An example code:

```
#include <malloc.h>
#include <stdlib.h>
#include <sys/mman.h>
```

[13]A detailed information about the pages mapped into the physical memory can be found in the `/proc/[PID]/smaps` file, where `[PID]` stands for the PID number of the process. Each mapping (i.e. part of the process address space) is provided with the following data:

`Size`	size of the mapping
`Rss` (Resident set size)	amount of the memory currently loaded into the physical memory (sum of the private and shared[14] memory)
`Pss` (Proportional set size)	= private + shared / number of the processes sharing the memory
`Shared Clean`	amount of the unmodified[15] shared memory
`Shared Dirty`	amount of the modified shared memory
`Private Clean`	amount of the unmodified private memory
`Private Dirty`	amount of the modified private memory

[14]The memory simultaneously accessed by several processes (e.g. libraries).

[15]The memory having an identical copy in the auxiliary memory storage.

```
void terminate(char* msg)
{
  perror(msg);
  exit(0);
}

void *allocate(int size)
{
  void *ptr = malloc(size);
  if(ptr == NULL) terminate("malloc() failed");
  return ptr;
}

int main(int argc, char* argv[])
{
  void *ptr1, *ptr2, ... , *ptrn;

  /* lock current and future memory used by the process */
  if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
    terminate("mlockall() failed");

  /* allocate all required memory */
  ptr1 = allocate(...);
  ptr2 = allocate(...);
         ...
  ptrn = allocate(...);

  /* allocated memory is locked, page fault cannot occur */

  return 0;
}
```

A heap of a sufficient size is locked at the beginning of the code in the second approach. Individual mappings outside the heap are not allowed. Therefore, the dynamic memory is allocated from a locked heap, so no page fault can occur. Of course, the maximum amount of the allocated memory at any runtime moment has to be estimated in advance. The memory losses due to fragmentation have to be considered. An example code:

```
#include <malloc.h>
#include <stdlib.h>
#include <sys/mman.h>

void terminate(char* msg, int pf)
{
  if(pf == 0) perror(msg);
  else printf(msg);
  exit(0);
}

int main(int argc, char* argv[])
{
  void *ptr;
```

```
  /* disable individual mappings outside the heap */
  int err = mallopt(M_MMAP_MAX, 0);
  /* disable decreasing program break (heap trimming)     */
  /* note: released but locked heap memory is unlocked at */
  /*       program break decrease                         */
  err = err + mallopt(M_TRIM_THRESHOLD, -1);
  if(err < 2) terminate("mallopt() failed\n", 1);

  /* lock current and future memory used by the process */
  if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
    terminate("mlockall() failed", 0);

  /* allocate and free a sufficient chunk of memory            */
  /* heap is increased to its final size (e.g.  100kB + padding) */
  ptr = malloc(100 * 1024);
  if(ptr == NULL) terminate("malloc() failed", 0);
  free(ptr);

  /* chunk of memory on heap is locked                          */
  /* page fault will not occur if program break is not increased */

  return 0;
}
```

Although the page faults are eliminated, the memory allocation functions (e.g. `malloc()`, `free()`, etc.) are however still unpredictable. They are not deterministic because of heap fragmentation. The time to allocate or free the memory space on a heap is neither constant nor known in advance.

Stack page faults. To avoid the page faults because of the stack data access, a sufficient chunk of memory has to be locked for the stack usage. This can be achieved by a simple trick. A page used for the stack is locked at the first access. It is never unlocked. To prevent the page faults because of stack growing, the stack should be completely used once at the beginning of the code. This is done by calling a dummy function (i.e. `touch_stack()`) with a large local array accessed once per every page. After return, the accessed stack space will be released, but locked. Of course, the required amount of the stack space has to be estimated in advance. An example code:

```
#include <unistd.h>
#include <sys/mman.h>
#include <stdlib.h>

/* touch 100kB on stack (every page is accessed once) */
void touch_stack()
{
  char stack_space[100 * 1024];
  int i, page_size = sysconf(_SC_PAGESIZE);
  for(i = 0; i < 100 * 1024; i = i + page_size) stack_space[i] = 0;
}

int main(int argc, char* argv[])
{
  /* lock current and future memory used by the process */
```

```
  if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
  {
    perror("mlockall() failed");
    exit(0);
  }

  /* lock stack */
  touch_stack();

  /* chunk of memory is locked for the stack                   */
  /* page fault will not occur if the stack does not grow over */
  /* the locked chunk                                          */

  return 0;
}
```

5.2.4 High-resolution timer

In the Linux kernel, the accuracy of the time operations (e.g. timeouts, CPU time measurements, etc.) is limited by a software clock. Its precision is typically in the range of milliseconds. A sub-millisecond resolution is often required in real-time applications. Since the kernel software clock is not accurate enough, a high-resolution timer has to be used. A high-resolution timer measures the time as accurately as the hardware allows. The hardware precision can be found out by the `clock_getres()` function. The `clock_gettime()` function is used to retrieve the high-resolution timer value, and the `clock_nanosleep()` function realizes the timeout with a sub-millisecond precision. The example code:

```
/* hrt_example.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void terminate(char *msg, int err)
{
  if(err == 0) perror(msg);
  else printf("%s failed: %s\n", msg, strerror(err));
  exit(0);
}

int main(int argc, char* argv[])
{
  struct timespec t;
  int err;

  /* get high resolution timer precision */
  if(clock_getres(CLOCK_MONOTONIC, &t) == -1)
    terminate("clock_getres() failed", 0);
  printf("HRT precision: %dns\ n", t.tv_sec * 1000000000 + t.tv_nsec);

  /* get current high resolution timer value */
  if(clock_gettime(CLOCK_MONOTONIC, &t) == -1)
```

```
    terminate("clock_gettime() failed", 0);
  printf("HRT value: %ds %dns\n", t.tv_sec, t.tv_nsec);

  /* sleep for 10000000100ns since clock_gettime() call */
  t.tv_sec = t.tv_sec + 10;
  t.tv_nsec = t.tv_nsec + 100;
  if(t.tv_nsec > 999999999)
  {
    t.tv_sec = t.tv_sec + 1;
    t.tv_nsec = t.tv_nsec - 1000000000;
  }
  err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
  if(err != 0) terminate("clock_nanosleep()", err);

  return 0;
}
```

To use high resolution timer functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` complier (e.g. `gcc -o hrt_example hrt_example.c -lrt`).

The sub-millisecond timeout can also be realized with a timer created by the `create_timer()` function. The timer is armed by the `timer_settime()` function. On expiration, the timer can do nothing, send a signal (see section 6.2), or start a new thread (see page 129). The timeout is carried out while the process waits for a blocked signal in the `sigwait()` function. The timer is deleted by the `delete_timer()` function. To employ timer functions, the `gcc` complier `-lrt` option has to be used. An example code:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>

void terminate(char *msg, int err)
{
  if(err == 0) perror(msg);
  else printf("%s failed: %s\n", msg, strerror(err));
  exit(0);
}

int main(int argc, char* argv[])
{
  sigset_t sset;
  struct sigevent ev;
  timer_t timer;
  struct itimerspec tset;
  int err, sig;

  /* block SIGUSR1 */
  if(sigemptyset(&sset) == -1) terminate("sigemptyset() failed", 0);
  if(sigaddset(&sset, SIGUSR1) == -1)
    terminate("sigaddset() failed", 0);
  if(sigprocmask(SIG_BLOCK, &sset, NULL) == -1)
    terminate("sigprocmask() failed", 0);
```

```
  /* create timer that sends SIGUSR1 on expiration */
  ev.sigev_notify = SIGEV_SIGNAL;
  ev.sigev_signo = SIGUSR1;
  ev.sigev_value.sival_ptr = &timer;
  if(timer_create(CLOCK_MONOTONIC, &ev, &timer) == -1)
    terminate("timer_create() failed", 0);

  /* get current time value */
  if(clock_gettime(CLOCK_MONOTONIC, &(tset.it_value)) == -1)
    terminate("clock_gettime() failed", 0);

  /* arm timer to start after 5s and expire every 10s */
  tset.it_value.tv_sec = tset.it_value.tv_sec + 5;
  tset.it_interval.tv_sec = 10;
  tset.it_interval.tv_nsec = 0;
  if(timer_settime(timer, TIMER_ABSTIME, &tset, NULL) == -1)
    terminate("timer_settime() failed", 0);

  /* sleep for initial 5s */
  err = sigwait(&sset, &sig);
  if(err != 0) terminate("sigwait()", err);

  /* sleep for another 10s */
  err = sigwait(&sset, &sig);
  if(err != 0) terminate("sigwait()", err);

  /* delete timer */
  if(timer_delete(timer) == -1)
    terminate("timer_delete() failed", 0);

  return 0;
}
```

When a signal is delivered to a process, it is handled by the process signal handler function. A blocked signal is not delivered. It waits to be unblocked as a pending signal. The `sigwait()` function suspends the process execution and waits for the pending (i.e. blocked) signal to arrive. On arrival, `sigwait()` returns, the process continues and the signal is removed from the pending signal list. Thus, the signal is actually never handled by the signal handler. For `sigwait()` and other signal functions, see section 6.2.

5.2.5 Real-time application skeleton

A real-time application should not miss its deadlines. The task has to be always performed on time. To keep the latencies as low as possible, a real-time pre-emptive kernel is required. The priority, scheduling policy and page-fault preventing technique have to be set in the application initialisation. An example skeleton code:

```
#include ...

void touch_stack() { ... }

int main(int argc, char* argv[])
```

```
{
  /* initialization */
  sched_setscheduler(...); /* priority and scheduling */
  mlockall(...);
  mallopt(...); malloc(...); free(...); /* heap */
  touch_stack(...); /* stack */

  clock_gettime(...); /* get current time value */
  while(1)
  {
    /* do the task */

    /* set next time limit according to current time */
        ...
    /* wait for next time limit */
    clock_nanosleep(...);
  }

  return 0; /* never reached */
}
```

After initializations, the application runs in the indefinite loop. It wakes up and performs the task on time, then sleeps until the next time limit. The `sigwait()` function can be used instead of `clock_nanosleep()`. In that case, the indefinite loop would look like:

```
  /* block signal */
  sigemptyset(...); sigaddset(...); sigprocmask(...);

  timer_create(...); /* create timer */
  clock_gettime(...); /* get current time value */
        ...
  timer_settime(...); /* arm timer */
  while(1)
  {
    /* do the task */

    /* wait for the next deadline */
    sigwait(...);
  }
```

Chapter 6

Inter-process communication[1]

Processes and threads can communicate with each other in several ways. A thread is a "subprocess" created inside a process. A process can contain multiple threads managed by a kernel like the independent processes. The treads existing inside a process share the resources, such as the address space, code and context (i.e. variables, etc.). Yet, each thread maintains its own stack, CPU state, scheduling policy and priority, set of pending and blocked signals, etc. Different processes do not share the resources, which is the difference between the two.

The inter-process and inter-thread communication is a technique to transfer data among different processes and threads. The treads and processes can communicate through signals, pipes, named pipes or FIFOs, message queues, shared-memory segments, memory-mapped files or sockets, to name some mechanisms available. Describing the inter-process communication mechanisms in detail exceeds the scope of this textbook. A few examples follow.

The inter-process communication mechanisms [50] can be divided into the asynchronous and synchronous ones. With the asynchronous mechanisms (e.g. signals, shared-memory segments and memory-mapped files), the receiver does not have any control when the data is delivered. On the other side, with the synchronous mechanisms (e.g. pipes, message queues and sockets), the receiver explicitly asks for the data. If the data is not available, the receiver can wait (blocking mode) or continue empty-handed (non-blocking mode)[2].

From the real-time perspective, one must be aware that inter-process communication in any form is not carried out instantly. Some delay is always present.

6.1 Creating/terminating threads and processes

6.1.1 Threads

A new thread inside a process can be created with the `pthread_create()` function. A thread is started by invoking a specified function. To use POSIX [51] (Portable Operating System Interface) thread functions, the `libpthread.a` library has to be linked. Use the `-lpthread` option with the `gcc` compiler. An example code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

[1]The functions used in this chapter are POSIX-compliant [51]. For a detailed explanation, one can also use [10, 11].

[2]Since the communication channel (i.e. pipe size) is limited, the sender may also wait or continue without sending.

```
void pthread_create_err(pthread_t *thread, const pthread_attr_t
                   *attr, void *(*start_routine)(void *), void *arg)
{
  int err = pthread_create(thread, attr, start_routine, arg);
  if(err == 0) return;
  printf("pthread_create() failed: %s\n", strerror(err));
  exit(0);
}

void *thread1(void *arg)
{

  thread process 1
  on error, use pthread_exit()  (the main and other threads are not
                                 terminated)
  on fatal error, use exit()  (the main and other threads are also terminated)

  return NULL;
}

void *thread2(void *arg)
{

  thread process 2
  on error, use pthread_exit()  (the main and other threads are not
                                 terminated)
  on fatal error, use exit()  (the main and other threads are also terminated)

  return NULL;
}


        ...

int main(int argc, char *argv[])
{
  pthread_t id1, id2, ...;

  /* start threads */
  pthread_create_err(&id1, NULL, thread1, NULL);
  pthread_create_err(&id2, NULL, thread2, NULL);
        ...

  main process
  to terminate a single thread, use pthread_cancel()
  on error, use exit()  (threads are terminated)

  return 0;
}
```

The `exit()` function is used for normal process termination. All the threads, including the main one, are terminated when one of the threads calls `exit()` or when the main thread ends. A single thread can exit with the `pthread_exit()` function or can be terminated with the `phtread_cancel()` function.

6.1.2 Processes

A child process can be created with the `fork()` function. It is an exact duplicate of the parent process. PID of the child process is returned to the parent and zero to the child. In the following example code, the `fork()` is wrapped into the `fork_child()` function:

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <errno.h>

void terminate(char *msg, int child)
{
  if(msg != NULL) perror(msg);

  /* terminate child process with error */
  if(child > 0) _exit(0);

  /* terminate child process with fatal error */
  if(child < 0) _exit(1);

  /* terminate children then parent */
  if(kill(0, SIGTERM) == -1) terminate("kill() failed", 0);
  exit(0);
}

/* SIGTERM handler */
void sigterm(int signo)
{
  /* exit child process */
  terminate(NULL, 1);
}

/* SIGCHLD handler */
void sigchld(int signo)
{
  /* handle terminated child processes */
  while(1)
  {
    int status;

    /* terminate zombie */
    pid_t pid = waitpid(-1, &status, WNOHANG);[3]

    if(pid == -1)
    {
      /* no children left */
      if(errno == ECHILD) break;

      terminate("wait() failed", 0);
```

[3]The `waitpid()` function returns immediately (`WNOHANG` option) with PID of the terminated child process, zero if there is none and -1 on an error (no children left is classified as an error). The terminated child in the zombie state is released.

```
    }

    /* no zombies left */
    if(pid == 0) break;

    /* child fatal error, terminate other children and parent */
    if(WIFEXITED(status) && WEXITSTATUS(status) != 0)
      terminate(NULL, 0);
  }
}

pid_t fork_child()
{
  pid_t pid = fork();
  if(pid == -1) terminate("fork() failed", 0);
  return pid;
}

int child1(sigset_t sset)
{
  /* unblock SIGTERM in child process */
  if(sigprocmask(SIG_UNBLOCK, &sset, NULL) == -1)
    terminate("sigprocmask() failed", 1);

  child process 1
  on error, use terminate(..., 1)
  on fatal error, use terminate(..., -1)

  return 0;
}

int child2(sigset_t sset)
{
  /* unblock SIGTERM in child process */
  if(sigprocmask(SIG_UNBLOCK, &sset, NULL) == -1)
    terminate("sigprocmask() failed", 1);

  child process 2
  on error, use terminate(..., 1)
  on fatal error, use terminate(..., -1)

  return 0;
}

        ...

int main(int argc, char *argv[])
{
  sigset_t sset;
  struct sigaction act;
  pid_t pid1, pid2, ...;

  /* block SIGTERM and define handler */
  if(sigemptyset(&sset) == -1)
```

```
    terminate("sigemptyset() failed", 0);
  act.sa_handler = sigterm;
  act.sa_mask = sset;
  act.sa_flags = 0;
  if(sigaddset(&sset, SIGTERM) == -1)
                              terminate("sigaddset() failed", 0);
  if(sigprocmask(SIG_BLOCK, &sset, NULL) == -1)
                            terminate("sigprocmask() failed", 0);
  if(sigaction(SIGTERM, &act, NULL) == -1)
                              terminate("sigaction() failed", 0);

  /* define SIGCHLD handler */
  act.sa_handler = sigchld;
  if(sigaction(SIGCHLD, &act, NULL) == -1)
                              terminate("sigaction() failed", 0);

  /* start child processes */
  pid1 = fork_child();
  if(pid1 == 0) return child1(sset);
  pid2 = fork_child();
  if(pid2 == 0) return child2(sset);
        ...
```

parent process
to terminate a single child, use `kill(pid, SIGTERM)`
on error, use `terminate(..., 0)`

```
  /* terminate children before return */
  if(kill(0, SIGTERM) == -1) terminate("kill() failed", 0);
  return 0;
}
```

Creating child processes is easy. The situation gets slightly more complicated when for some reason all child processes and parent process have to be terminated (e.g. because of a fatal error in one of the child processes).

The terminating mechanism is controlled by two signals, i.e. `SIGTERM` and `SIGCHLD`. The `sigterm()` and `sigchld()` signal handler functions are defined for both signals (see section 6.2) and `SIGTERM` is blocked. Therefore, the `SIGTERM` handler should never be called in the parent process. At child creation, `SIGTERM` is unblocked. Thus, the `SIGTERM` handler can be called in the child processes.

On child completion, `SIGCHLD` is sent to the parent while the child remains as a zombie process (see page 24). The signal is also sent if the `_exit()` function is used. So the parent is noticed about the child termination, the parent `SIGCHLD` handler `sigchld()` is called. The handler terminates the child zombie process in `waitpid()` and further decides whether the child termination was fatal and all other children and the parent should also be terminated. The parent terminates all its children by sending `SIGTERM` with the `kill()` function. The mechanism is shown in Fig. 6.1. The `while` loop in `sigchld()` ensures that all zombies are terminated when more than one are waiting (e.g. if two children terminate at the same time, the parent receives only one `SIGCHLD`).

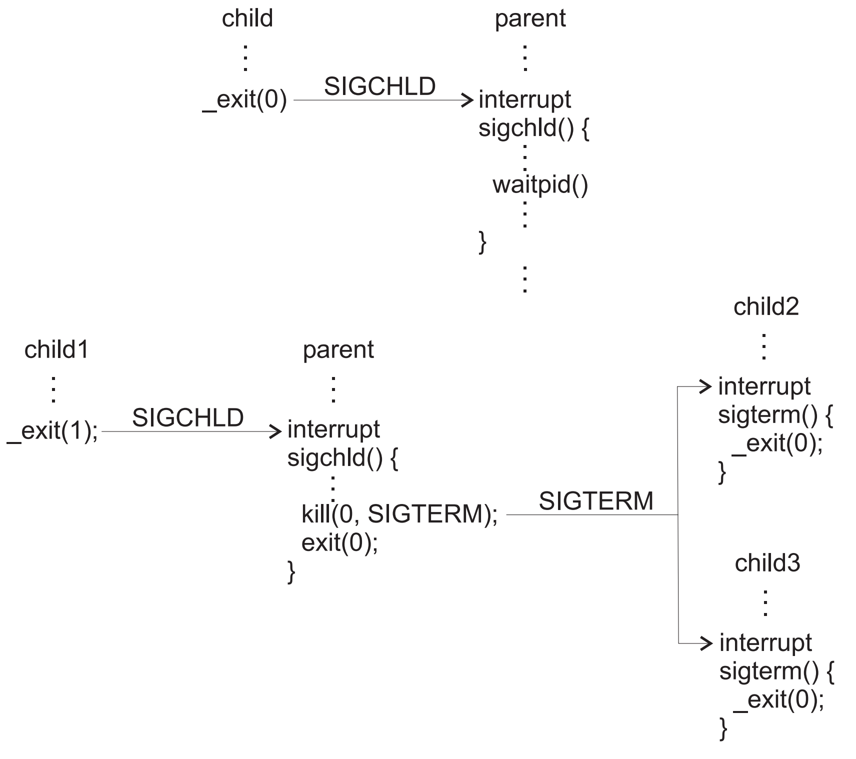

Figure 6.1: Child process normal (above) and fatal (below) termination

6.2 Signals

When a signal (see page 26) is delivered to a process, the process is immediately interrupted. The default action specified for the signal (i.e. terminate, stop or continue the process, ignore the signal, etc.) takes place. A signal is sent to a process with the `kill()`[4] function, which returns zero on success, -1 otherwise. Its declaration is in the `signal.h` header file. A signal can be sent to an arbitrary process identified by PID.

`int error = kill(pid,` *signal_number*`);`

Instead of performing a default action on the signal arrival, a signal can be caught by the signal handler function. The signal handler is established by the `sigemptyset()` and `sigaction()` functions. `sigemptyset()` creates an empty set of signals used as a set of signals temporary blocked during handler execution, and `sigaction()` defines the handler. Both functions return zero on success, -1 otherwise. Their declarations are in the `signal.h` header file. Only the signal-safe functions have to be used in the signal handler[5].

[4]When a signal is sent to a thread in the same process, the `pthread_kill()` function has to be used instead of `kill()`. When a signal is sent to a multithreaded process, it is delivered to one of threads which does not block the signal. It is unspecified which thread gets the signal since the signals sent by `kill()` are process-directed.

[5]A signal-safe or reentrant function can be interrupted and safely called again before the original call completes. Reentrancy is required for all functions used in the main program and signal handler, since the process can be interrupted by the signal handler at any point, e.g. in the middle of a function call.

```
void sighandler(int signo) {...}
        ...
struct sigaction act;
act.sa_handler = sighandler;
int error = sigemptyset(&act.sa_mask);
act.sa_flags = 0;
error = sigaction(signal_number, &act, NULL);
```

A signal may be blocked. A blocked signal is not delivered until unblocked. In the meanwhile, the signal is pending. The `sigemptyset()`, `sigaddset()` and `sigprocmask()` functions are used to define the blocked signals. `sigemptyset()` creates an empty set of signals, `sigaddset()` adds a signal to the set and `sigprocmask()`[6] sets a mask of the blocked signals for the process. The functions return zero on success, -1 otherwise. Their declarations are in the `signal.h` header file.

```
sigset_t sset;
int error = sigemptyset(&sset);
error = sigaddset(&sset, signal_number);
error = sigprocmask(SIG_BLOCK, &sset, NULL);
```

A blocked signal does not interrupt the process. A process can wait for a blocked signal to arrive, or check if there are any signals pending (i.e. already arrived blocked signals). That indicates that signals can be accepted synchronously. Thus, the process can decide when the signal will be handled. Waiting for a blocked signal arrival is implemented in the `sigwait()` function. The function waits for any blocked signal specified in a given signal set. It returns when a blocked signal (from the set) arrives. If there is at least one signal (from the set) pending, `sigwait()` returns immediately. The accepted signal is removed from the list of pending signals, its number is returned in `sig_no`. The `sigwait()` returns zero on success, an error number otherwise. Its declaration is in the `signal.h` header file.

```
int error, sig_no;
/* sset is subset of set of blocked signals */
error = sigwait(&sset, &sig_no);
```

6.3 Pipes and named pipes

Pipes are used for a serial unidirectional communication. The data is sent on the pipe write end and received on its read end. The data order is always preserved (first in first out). Regular pipes can be used for communication between two threads of the same process or between two related processes (i.e. two children of the same parent process, or a child and parent process). A pipe is created with the `pipe()` function. The read and write end file descriptors are returned in `fd[0]` and `fd[1]`. A file descriptor is closed by the `close()` function. Both functions return zero on success, -1 otherwise. The declarations are in the `unistd.h` header file.

```
int error, fd[2];
error = pipe(fd);
```

[6] Use the `pthread_sigmask()` function instead of `sigprocmask()` in a multithreaded process to set a mask of the blocked signals for a single thread.

```
        ...
error = close(fd[0]);
error = close(fd[1]);
```

A file descriptor created in one thread can be used in another since the threads in a multithreaded process share resources. When a file descriptor is closed, it is closed for all threads. A different situation occurs with the parent and child processes when a pipe is created before the child is forked. The child inherits copies of the file descriptors. Thus the parent and child both have their own copies of the read and write end file descriptors of the same pipe. The unused file descriptors should be closed (e.g. when the parent writes to the pipe and the child reads from it, the parent read end and the child write end file descriptors should be closed).

A named pipe has a name in the file system. Thus, it can be used for communication between two unrelated processes. The named pipe is created with `mkfifo()` and destroyed by the `unlink()` function. Both functions return zero on success, -1 otherwise. Declarations are in the `unistd.h` header file. The named pipe must have the read and write access rights (see section 1.5) in order to enable writing to and reading from the pipe. The access rights are given in the second argument of `mkfifo()`. A file descriptor to the named pipe is created with the `open()` function which returns -1 on an error. Its declaration is in the `fcntl.h` header file.

```
int error, fd[2];
error = mkfifo("named pipe filename", 0600);          /* rw------- */
fd[0] = open("named pipe filename", O_RDONLY);
fd[1] = open("named pipe filename", O_WRONLY);
        ...
error = close(fd[0]);
error = close(fd[1]);
error = unlink("named pipe filename");
```

Once the file descriptors are available, writing to and reading from a regular or named pipe can be performed with the standard `write()` and `read()` functions. Both functions return the number of bytes written or read, -1 on an error. Declarations are in the `unistd.h` header file.

```
char buffer[SIZE];
int num_of_bytes = write(fd[1], buffer, SIZE);
num_of_bytes = read(fd[0], buffer, SIZE);
```

The capacity is the maximum number of bytes a pipe can hold. The default capacity on Linux is 64kB. When a pipe is full, no additional bytes can be written until some are read out.

A pipe is opened in the blocking mode by default. Therefore, `read()` blocks until at least one byte is available. Similarly, `write()` blocks until enough space is available. Blocking automatically synchronizes the write and read end of the pipe. To make `read()` and `write()` return immediately, the non-blocking mode has to be specified. The non-blocking mode can be specified with the `O_NONBLOCK` flag in the `open()` function. Or, it can be specified later by the `fcntl()` function. `fcntl()` returns -1 on an error. Its declaration is in the `fcntl.h` header file.

```
file descriptor = open("filename", other flags | O_NONBLOCK);
```

... or, if file descriptor was opened without `O_NONBLOCK` *...*

```
int error = fcntl(file descriptor, F_SETFL, O_NONBLOCK);
```

Writing to a pipe with the `write()` function is an atomic operation of up to `PIPE_BUF` bytes (e.g. 4kB on Linux). `PIPE_BUF` is a kernel-dependent predefined constant defined in the `linux/limits.h` header file. It cannot be altered. It is guaranteed that a write request up to the `PIPE_BUF` bytes is written in a contiguous sequence. Larger requests may be interleaved with the data written to the same pipe by other threads or processes.

A read request to a pipe with no open write end file descriptor returns immediately with the zero bytes read (i.e. `EOF`). A write request to a pipe with no open read end file descriptor sends the `SIGPIPE` signal which by default terminates the process.

6.4 Message queues

A message is an indivisible block of data sent and received as a whole. Processes can exchange messages through message queues. Messages have priority. The message with the highest priority is delivered first. If there are several messages with the same priority, the oldest is delivered first. The message queue has a name[7] in the virtual file system[8]. Therefore, it can be used by unrelated processes. A message queue is created with `mq_open()` and destroyed by the `mq_unlink()` function. To create a new message queue, the `O_CREAT` flag and access rights have to be specified. Without `O_CREAT`, `mq_open()` opens the already existing queue. The message queue descriptor is closed by the `mq_close()` function. All functions return -1 on an error. Their declarations are in the `mqueue.h` header file. To use the message queue functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` complier.

```
                                                      /* rw------- */
mqd_t mqdw = mq_open("/mqname", O_CREAT | O_WRONLY, 0600, NULL);
mqd_t mqdr = mq_open("/mqname", O_RDONLY);
        ...
int error = mq_close(mqdr);
error = mq_close(mqdw);
error = mq_unlink("/mqname");
```

Once the message queue descriptor is available, writing to it can be performed by the `mq_send()` function, which returns zero on success. Reading from the message queue is done with the `mq_receive()` function. The received message buffer size must be large enough to hold the longest message possible (e.g. 8kB). The maximum message length[9] can be obtained by the `mq_getattr()` function, which returns zero on success. All functions return -1 on an error. Declarations are in the `mqueue.h` header file.

```
struct mq_attr attrw, attrr;
int error = mq_getattr(mqdw, &attrw);
error = mq_getattr(mqdr, &attrr);
        ...
char *bufw = malloc(attrw.mq_msgsize);
```

[7]The message queue name is preceded by `/` character (e.g. `/mq1`).

[8]A virtual file system with message queues can be mounted by a super user with the `mount -t mqueue none` *mounting point* command (see page 10).

[9]And other message queue properties like the mode (blocking/non-blocking), number of messages in a queue and maximum number of messages in queue.

```
char *bufr = malloc(attrr.mq_msgsize);
        ...
error = mq_send(mqdw, bufw, attrw.mq_msgsize, priority);
        ...
int priority, num_of_bytes;
num_of_bytes = mq_receive(mqdr, bufr, attrr.mq_msgsize, &priority);
        ...
free(bufr);
free(bufw);
```

The message queue descriptor is opened in the blocking mode by default. Therefore, `mq_receive()` blocks until the message is available. Similarly, `mq_send()` blocks if the queue is full. To make `mq_receive()` and `mq_send()` return immediately, the non-blocking mode has to be specified. The non-blocking mode can be specified with the `O_NONBLOCK` flag in the `mq_open()` function. Or, it can be specified later by the `mq_setattr()` function, which returns zero on success, -1 otherwise. Its declaration is in the `mqueue.h` header file.

```
mqdescriptor = open("/mqname", other flags | O_NONBLOCK , ...);
```

... or, if message queue descriptor was opened without `O_NONBLOCK` *...*

```
int error;
struct mq_attr attr;
attr.mq_flags = O_NONBLOCK;
int error = mq_setattr(mqdescriptor, &attr, NULL);
```

An asynchronous notification on the message arrival can be enabled/disabled with the `mq_notify()` function. It is disabled by default. With the asynchronous notification enabled, either a signal is sent or a new thread is started on the message arrival. The process can immediately react in the signal handler or thread function.

6.5 Shared-memory segments

A shared memory provides the simplest and the fastest form of an inter-process communication. The same memory segment is shared among the unrelated processes. Access to a shared memory is by all means equivalent to a privately-allocated memory (e.g. with `malloc()`) access. A modification done by one process is instantly seen to all the others. Since there is no synchronization provided, a race conditions[10] can occur (e.g. two processes write to the same location at the same time, or one process reads data before it is actually written by another, etc.).

A shared memory segment has a name[11] in the virtual file system[12]. Therefore, it can be used by unrelated processes. A shared-memory segment is created with the `shm_open()` and destroyed by the `shm_unlink()` function. To create a new shared-memory segment, the `O_CREAT` flag and access rights have to be specified. Without `O_CREAT`, `shm_open()` opens the already existing memory segment, the third argument with the access rights is ignored. Both functions return -1 on an error. Their declarations are in the `fcntl.h` header file. To use the shared-memory

[10] A race condition occurs when the output depends on a sequence or timing of outside events.
[11] The shared memory segment name is preceded by / character (e.g. `/shm1`).
[12] A virtual file system of the `tmpfs` type mounted on `/dev/shm`.

segment functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` complier. The shared-memory segment descriptor is closed with the standard `close()` function (see page 135).

The default size of a newly-created shared-memory segment is zero bytes. The segment size is defined by the `ftruncate()` function. To define the segment size, the segment must be opened for writing (e.g. with `O_RDWR` flag). `ftruncate()` returns zero on success, -1 otherwise. Its declaration is in the `unistd.h` header file.

```
                                                /* rw------- */
int shmdw = shm_open("/shmname", O_CREAT | O_RDWR, 0600);
int error = ftruncate(shmdw, size in bytes);
int shmdr = shm_open("/shmname", O_RDONLY, ignored);
        ...
error = close(shmdr);
error = close(shmdw);
error = shm_unlink("/shmname");
```

The pointer to the shared memory is obtained from the segment descriptor with the `mmap()` function. The function maps a file into the process address space. The memory protection mode is passed in the third argument and the fourth argument defines visibility of the shared-memory updates to other processes. Of course, the protection mode must not conflict with the opening mode[13] and visibility to other processes has to be granted to share the segment. Once the pointer is obtained, the shared memory can be accessed. Mapping is omitted by the `munmap()` function. Both functions return -1 on an error. Their declarations are in the `sys/mman.h` header file.

```
void *ptrw = mmap(NULL, size in bytes, PROT_WRITE, MAP_SHARED, shmdw, 0);
void *ptrr = mmap(NULL, size in bytes, PROT_READ,  MAP_SHARED, shmdr, 0);
        ...
int error = munmap(ptrr, size in bytes);
error = munmap(ptrw, size in bytes);
```

6.6 Memory-mapped files

A file can be mapped into the process address space. An entire file is copied into the memory. Its contents can be read or modified as any other allocated memory (e.g. with `malloc()`). The changes made are automatically written back to the file. If modifications are written back immediately, they can be instantly seen by other unrelated processes mapping the same file. Therefore, a memory-mapped file can be used for the inter-process communication.

The mechanism is very similar as with the shared-memory segment although communication through a mapped file is slower since a true file is involved. A new file is created with `open()` with the `O_CREAT` flag and the access rights are specified. Without `O_CREAT`, `open()` opens the already existing file. The opened file is closed with `close()` and destroyed by the `unlink()` function (see page 135).

Before the file is mapped into the memory, it must be ensured that the file is large enough. The default size of a newly created file is zero bytes. The file length can be set by writing a dummy byte at the end position. The `lseek()` and

[13]The segment must be opened for reading (e.g. with `O_RDWR` flag) for `mmap()` to succeed, even when the memory protection mode allows only writing (e.g. `PROT_WRITE`).

`write()` functions (see page 136) are used. `lseek()` returns zero, -1 on an error. Its declaration is in the `unistd.h` header file.

```
                                                  /* rw------- */
int fdr, fdw = open("filename", O_CREAT | O_RDWR, 0600);
off_t offset = lseek(fdw, size in bytes - 1, SEEK_SET);
int num_of_bytes = write(fdw, "", 1);
offset = lseek(fdw, 0, SEEK_SET);
fdr = open("filename", O_RDONLY);
         ...
int error = close(fdr);
error = close(fdw);
error = unlink("filename");
```

A file is mapped into the memory with the `mmap()` function and unmapped by `munmap()` (see page 139).

6.7 Sockets

A socket is a communication endpoint represented by the file descriptor. Communication through sockets is the most flexible inter-process communication technique. Besides the regular inter-process communication among the processes running on the same host machine, the sockets enable a communication among the processes running on different hosts (see subsection 4.4.6). A client-server model is used. The server process waits for the client to initiate the communication. The server normally answers to the client request.

The communication through a socket is defined by the style (connection-oriented or connectionless), domain (e.g. local (the socket address is the filename), Internet (the socket address is the IP address and port number), etc.) and protocol (e.g. Unix domain protocol, TCP, UDP, etc.). All combinations of the styles, domains and protocols are not supported. The socket communication in the Internet domain is explained in subsection 4.4.6. This section focuses on the local domain.

The server and client code in the local domain is very similar to the code in the Internet domain in subsection 4.4.6. Only slight modifications are required. In contrast to the Internet domain, in the local domain, the data delivery on a connectionless socket is as reliable as on a connection-oriented socket. Reordering never takes place. However, packets are discharged on a connectionless socket when its buffer is full.

Connection-oriented local-domain sockets

The following lines in the C programming language represent a connection-oriented server in the local domain. `AF_UNIX` and `sockaddr_un` are used instead of `AF_INET` and `sockaddr_in` regarding the code on page 107. Note that the socket path (*filename*) is relative and that the `unlink()` function at the end removes the socket from the file system.

```
int socketfd, error, size, connectionfd, num_of_bytes;
struct sockaddr_un server, client;
char buffer[SIZE];
    ...
socketfd = socket(AF_UNIX, SOCK_STREAM, 0);
memset(&server, 0, sizeof(struct sockaddr_un));
```

```
server.sun_family = AF_UNIX;
strcpy(server.sun_path, "filename");
error = bind(socketfd, (struct sockaddr *)&server,
                                        sizeof(struct sockaddr_un));
     ...
error = listen(socketfd, NUM_OF_CONN);
size = sizeof(struct sockaddr_un);
connectionfd = accept(socketfd, (struct sockaddr *)&client, &size);
     ...
num_of_bytes = read(connectionfd, buffer, SIZE);
     ...
num_of_bytes = write(connectionfd, buffer, SIZE);
     ...
error = close(connectionfd);
error = close(socketfd);
error = unlink("filename");
```

The connection-oriented client code derived from the code on page 109 is as expected.

```
int descriptor, error, num_of_bytes;
struct sockaddr_un dest;
char buffer[SIZE];
     ...
descriptor = socket(AF_UNIX, SOCK_STREAM, 0);
memset(&dest, 0, sizeof(struct sockaddr_un));
dest.sun_family = AF_UNIX;
strcpy(dest.sun_path, "filename");
error = connect(descriptor, (struct sockaddr *)&dest,
                                        sizeof(struct sockaddr_un));
     ...
num_of_bytes = write(descriptor, buffer, SIZE);
     ...
num_of_bytes = read(descriptor, buffer, SIZE);
     ...
error = close(descriptor);
```

The blocking avoiding techniques described on pages from 109 to 111 can be applied in the local domain. The `fork()`, `select()` and `fcntl()` functions can be used in the same way as in the Internet domain.

Connectionless local-domain sockets

The following lines in the C programming language represent a connectionless server in the local domain. The code is similar to the code in the Internet domain (see page 111) with the local-domain specialties (`AF_UNIX`, `sockaddr_un`, `unlink()`).

```
int socketfd, error, size, num;
struct sockaddr_un server, client;
char buffer[SIZE];
     ...
socketfd = socket(AF_UNIX, SOCK_DGRAM, 0);
memset(&server, 0, sizeof(struct sockaddr_un));
```

```
server.sun_family = AF_UNIX;
strcpy(server.sun_path, "server socket filename");
error = bind(socketfd, (struct sockaddr *)&server,
                                        sizeof(struct sockaddr_un));
    ...
size = sizeof(struct sockaddr_un);
num = recvfrom(socketfd, buffer, SIZE, 0,
                                      (struct sockaddr *)&client, &size);
    ...
num = sendto(socketfd, buffer, SIZE, 0, (struct sockaddr *)&client,
                                        sizeof(struct sockaddr_un));
    ...
error = close(socketfd);
error = unlink("server socket filename");
```

A connectionless client code in the local domain follows. The socket path name is not automatically assigned in the Unix-domain protocol. The client must bind[14] its socket to some file to define the return path for the server. Otherwise, the server cannot answer. Thus, there are two socket files, the server's and the client's. The client's code is in fact the same as the server's.

```
int socketfd, error, size, num;
struct sockaddr_un client, dest;
char buffer[SIZE];
    ...
socketfd = socket(AF_UNIX, SOCK_DGRAM, 0);
memset(&client, 0, sizeof(struct sockaddr_un));
client.sun_family = AF_UNIX;
strcpy(client.sun_path, "client socket filename");
error = bind(socketfd, (struct sockaddr *)&client,
                                        sizeof(struct sockaddr_un));
memset(&dest, 0, sizeof(struct sockaddr_un));
dest.sun_family = AF_UNIX;
strcpy(dest.sun_path, "server socket filename");
    ...
num = sendto(socketfd, buffer, SIZE, 0, (struct sockaddr *)&dest,
                                        sizeof(struct sockaddr_un));
    ...
size = sizeof(struct sockaddr_un);
num = recvfrom(socketfd, buffer, SIZE, 0, (struct sockaddr *)&dest,
                                                                &size);
    ...
error = close(socketfd);
error = unlink("client socket filename");
```

Abstract sockets

A regular Unix-domain socket is a file in the file system. An abstract Unix-domain socket is equivalent to the regular one except it does not exist in the file system. The socket is abstract if the first byte in its path is zero. In this section, the socket paths in the code (connection-oriented and connectionless, server and client) should be given by:

[14] Binding on the client side is not required in the Internet domain (see page 112).

strcpy(*sockaddr_ un structure*.sun_path + 1, "*abstract socket name*");

Since the abstract socket does not exist in the file system, `unlink()` is not required after closing. But, it can be used for communication between two unrelated processes though.

Socket pair

A socket pair provides a similar functionality as a pair of regular pipes (see section 6.3). One regular pipe provides a unidirectional communication. For a bidirectional communication, two pipes or a pair of sockets are/is required. A pair of connected sockets represents a bidirectional communication channel. The data written on one socket can be read on the other and vice versa.

A socket pair is created by the `socketpair()` function, which returns zero on success, -1 otherwise. Its declaration is in the `sys/socket.h` header file. The socket pairs can be created only in the Unix-domain protocol.

```
int error, sfd[2], num_of_bytes;
char buffer[SIZE];
     ...
error = socketpair(AF_UNIX, SOCK_STREAM or SOCK_DGRAM, 0, sfd);
     ...
num_of_bytes = write(sfd[0], buffer, SIZE);
num_of_bytes = read(sfd[1], buffer, SIZE);
     ...
or in the other direction
num_of_bytes = write(sfd[1], buffer, SIZE);
num_of_bytes = read(sfd[0], buffer, SIZE);
     ...
error = close(sfd[0]);
error = close(sfd[1]);
```

The sockets obtained by `socketpair()` do not have a name in the file system. Thus `unlink()` is not required after closing. A socket pair cannot be used for communication between two unrelated processes.

Chapter 7

Resource sharing and synchronization[1]

When two or more processes or threads use the same resource at the same time, a race condition can occur. The final result is not deterministic and depends on a sequence of events, which depends on timing.

Example: Processes A and B share global variable i located in a shared-memory segment. Both processes at some point increment the variable. Incrementing is performed in three steps: read the current value from the shared memory, increment the value and write the updated value back. If both processes try to increment variable i at the same time, a wrong result can be obtained:

event	value of i in shared memory	
...	N	
A reads i	N	
A increments i	N	
A is pre-empted by B	N	
B reads i	N	
B increments i	N	
B writes i back	$N+1$	
B stops, A continues	$N+1$	
A writes i back	$N+1$	(should be $N+2$)
...		

A resource-access control is required to avoid race conditions. It is provided by semaphores and mutexes.

7.1 Semaphore

A counting semaphore is an abstract-initialized kernel variable which cannot be less than zero [52]. It can be decremented or locked by a wait system call or incremented or unlocked by post system call. A wait call on semaphore S is denoted by $P(S)$and a post call by $V(S)$. Normally, a semaphore holds the current number of the available units of a particular resource. The initial value is the total number of the resource units. The process the locks corresponding semaphore before using the resource and unlocks it after. If the semaphore value is zero, it cannot be locked. The kernel blocks the process until some other process unlocks the semaphore. A simultaneous resource-access problem can be solved using semaphore S:

[1]The functions used in this chapter are POSIX-compliant [51]. For a detailed explanation one can also use [10, 11].

event	value of S	value of i in a shared memory
...	1	N
A calls $P(S)$	0	N
A reads i	0	N
A increments i	0	N
A is pre-empted by B	0	N
B calls $P(S)$	0	N
kernel blocks B, A continues	0	N
A writes i back	0	$N+1$
A calls $V(S)$	1	$N+1$
kernel unblocks B	1	$N+1$
A is blocked, B continues	1	$N+1$
B's $P(S)$ call continues	0	$N+1$
B reads i	0	$N+1$
B increments i	0	$N+1$
B writes i back	0	$N+2$
B calls $V(S)$	1	$N+2$
B stops, A continues	1	$N+2$
...		

If the semaphore initial value is one, as in the example above, the semaphore is named a binary semaphore. If the total number of the resource units is greater than one (e.g. the number of CPUs), then the corresponding counting semaphore has the information of how many units are available, but not which. An additional mechanism is required to locate the free units.

A semaphore is created/initialized with the `sem_init()` and destroyed by the `sem_destroy()` function. Both functions return zero on success, -1 otherwise. The declarations are in the `semaphore.h` header file. Only one process creates/initializes the semaphore, others just use it. To use the semaphore functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` compiler.

```
sem_t semaphore;
sem_t *semptr = &semaphore;
int error = sem_init(semptr, SHARED, initial value);
        ...
error = sem_destroy(semptr);
```

If a semaphore is used by related processes (i.e. threads), then `SHARED`=`0`. If a semaphore is used by unrelated processes, it must be created/initialized in the shared memory segment with `SHARED`=`1`. A semaphore created/initialized with `sem_init()` does not have a name in the file system.

A semaphore with a name[2] in a virtual file system is created with the `sem_open()` and destroyed by the `sem_unlink()` function. To create a new semaphore, the `O_CREAT` flag, access rights and initial value have to be specified. A read and write access should be granted. Without `O_CREAT`, `sem_open()` opens the already existing semaphore. Only one process creates the semaphore, others just open it. An opened semaphore is closed by the `sem_close()` function. All functions return -1 on an error. Their declarations are in the `semaphore.h` header file.

[2]A semaphore is created in a virtual file system of the `tmpfs` type mounted on `/dev/shm`. Its name is `sem.`*name*. The name passed in the `sem_open()` function is preceded by `/` character (e.g. `/`*name*).

To use the semaphore functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` complier.

```
                                                /* rw------- */
sem_t *semptr1 = sem_open("/name, O_CREAT, 0600, initial value);
sem_t *semptr2 = sem_open("/name, 0);
        ...
int error = sem_close(semptr2);
error = sem_close(semptr1);
error = sem_unlink("/name);
```

Once a semaphore is initialized, the wait and post system calls are performed by the `sem_wait()` and `sem_post()` functions. Both functions return zero on success, -1 otherwise. Their declarations are in the `semaphore.h` header file. Also use the `-lrt` option with the `gcc` complier.

```
int error = sem_wait(semptr);
error = sem_post(semptr);
```

7.1.1 Recursive deadlock

A recursive deadlock occurs when a process locks the semaphore several times, until its value is zero, before unlocking it. The process is therefore blocked and waits indefinitely for itself to unlock the semaphore (see Fig. 7.1). This typically happens in recursive functions. A recursive deadlock can be dealt with by using a mutex instead of a semaphore (see subsection 7.2.1).

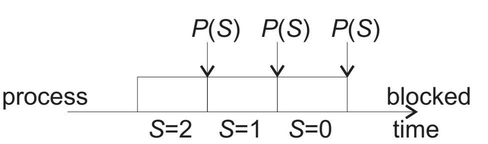

Figure 7.1: Recursive deadlock

7.1.2 Deadlock because of process termination

A process locks the semaphore. Then one or more other processes try to lock the same semaphore. They are blocked and are waiting for unlock. If the first process terminates for any reason before it unlocks the semaphore, the processes waiting for unlock are blocked indefinitely (see Fig. 7.2). A deadlock because of the process termination can be dealt with by using a mutex instead of a semaphore (see subsection 7.2.1).

7.1.3 Circular deadlock

A circular deadlock occurs when two or more processes develop circular semaphore locking. The first process locks semaphore S_1 and then waits for semaphore S_2, which was locked by the second process, which waits for semaphore S_3, which was locked by the third process, which waits for semaphore S_4, ..., which was locked by the n-th process, which waits for semaphore S_1. A two process circular deadlock is shown in Fig. 7.3.

A circular deadlock can be avoided by semaphore ordering. When a process wants to lock two or more semaphores at the same time, the locking must be

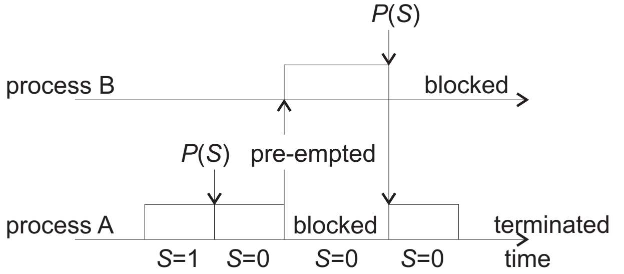

Figure 7.2: Deadlock because of process termination

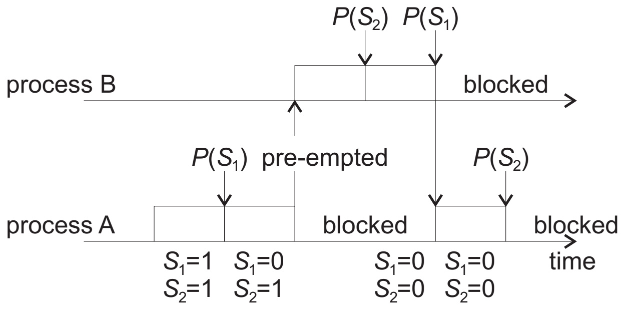

Figure 7.3: Circular deadlock

performed in order. In the example above, each process locked two semaphores in order (i.e. i-th process locked S_i first, then tried to lock S_{i+1}), except for the n-th process, which locked the two semaphores disorderly (i.e. S_n first, S_1 after). If the n-th process had stuck to the order, a circular deadlock would not have happen.

7.1.4 A priority-inversion problem

A priority-inversion problem occurs in real-time scheduling policies, where a priority takes precedence. It is only in one case that a high-priority process is forced to wait for a low-priority one. A low-priority process for instance locks the semaphore and is afterward pre-empted by a high-priority process which tries to lock the same semaphore. Since the semaphore is locked, the high-priority process is blocked until the low-priority process unlocks the semaphore. The problem arises if the third, a medium-priority process, pre-empts the low-priority process before the latter manages to unlock the semaphore[3]. Thus the high-priority process is in effect blocked by the medium-priority process for an undefined amount of time (Fig. 7.4).

A semaphore cannot deal with the priority-inversion problem. The priority-inversion prevention protocol on the mutex must be used (see pages 151 and 152).

7.2 MUTual EXclusion (mutex)

A mutex is a binary semaphore with ownership [53]. Since it is binary, it has two states: locked and unlocked. But unlike the binary semaphore, a locked mutex is owned. This means that only the process locking the mutex can unlock it. A simple principle of the ownership solves most of the semaphore problems.

[3]A single processor system is presumed.

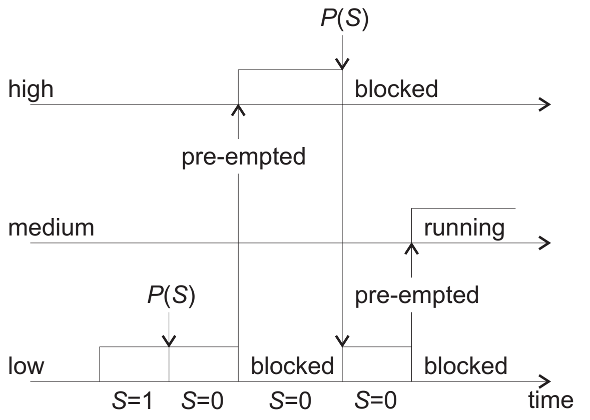

Figure 7.4: Priority inversion

A mutex with default attributes is created/initialized with `pthread_mutex_init()` and destroyed by the `pthread_mutex_destroy()` function. A mutex has to be unlocked when `pthread_mutex_destroy()` is called. Both functions return zero on success and an error number otherwise. The declarations are in the `pthread.h` header file. Only one thread creates/initializes the mutex, others just use it. A mutex does not have a name in the file system. Thus, it can be used by related processes/threads. If a mutex is used by unrelated processes, it must be created/initialized in a shared-memory segment and it must have the `PTHREAD_PROCESS_SHARED` attribute (see subsection 7.2.1). To use the mutex functions, the `librt.a` library has to be linked. Use the `-lrt` option with the `gcc` compiler.

```
pthread_mutex_t mutex;
int error = pthread_mutex_init(&mutex, NULL);
        ...
error = pthread_mutex_destroy(&mutex);
```

The second argument in `pthread_mutex_init()` defines the mutex attributes. `NULL` stands for the default attribute values. A mutex with default attributes does not check for errors. For instance, the ownership is not checked. Thus the ownership error, when a process not owning the mutex unlocks it, is not reported, which can lead to an undefined behavior. The same applies to other errors as well (i.e. unlocking an unlocked mutex, etc.). A default mutex without error checking is fast, but has to be used carefully.

Once a mutex is initialized, locking and unlocking are performed by the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. Both functions return zero on success and an error number otherwise. Their declarations are in the `pthread.h` header file. Also use the `-lrt` option with the `gcc` complier.

```
int error = pthread_mutex_lock(&mutex);
error = pthread_mutex_unlock(&mutex);
```

7.2.1 Mutex attributes

The non-default attribute values are assigned to a mutex at initialization by passing the `pthread_mutexattr_t` structure. Before usage, the structure must be initialized to the default values with `pthread_mutexattr_init()` and can be destroyed afterwards with the `pthread_mutexattr_destroy()` function. Both functions return zero on success and an error number otherwise. Their declarations are in the `pthread.h` header file. Use the `-lrt` option with the `gcc` complier.

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;
int error = pthread_mutexattr_init(&attr);
        ...
error = pthread_mutex_init(&mutex, &attr);
error = pthread_mutexattr_destroy(&attr);
```

The non-default attribute values are assigned with various functions. All of them return zero on success and an error number otherwise. Their declarations are in the `pthread.h` header file[4] and the `-lrt` option has to be used with the `gcc` complier.

If the mutex is to be used by unrelated processes, the `PTHREAD_PROCESS_SHARED` attribute is requested. The attribute is set by the `pthread_mutexattr_setpshared()` function.

```
err = pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
```

Error checking on the mutex lock and unlock operations is activated by the `PTHREAD_MUTEX_ERRORCHECK` attribute. The errors like `EDEADLK` (mutex is already locked) or `EPERM`[5] (mutex is locked by another process), etc., are returned by `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The `EDEADLK` error is in fact a recursive deadlock (see subsection 7.1.1). Recursion can be addressed due to the mutex ownership. The process locking the mutex owns it. So, the process can lock the same mutex again. Each lock increases the internal counter. Thus, unlocking has to be performed the same number of times to finally unlock the mutex. A recursive locking is enabled by the `PTHREAD_MUTEX_RECURSIVE` attribute. Both attributes can be set by the `pthread_mutexattr_settype()` function.

```
err = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
err = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
```

The deadlock because of the process termination (see subsection 7.1.2) is addressed by the `PTHREAD_MUTEX_ROBUST` attribute set with the `pthread_mutexattr_setrobust_np()`[6] function. Because of the ownership, the kernel knows which mutexes are owned/locked by the terminated process. The kernel unlocks those mutexes, but it also marks their state as inconsistent. When another process tries to lock such a mutex, `pthread_mutex_lock()` locks the mutex and returns the `EOWNERDEAD`[5] error number indicating the mutex inconsistent state. The resource protected by the mutex may be corrupted. If and when the resource is recovered, the `pthread_mutex_consistent_np()`[6] function should be called.

---

[4] Some of the following mutex attributes are part of XSI (X/Open System Interface) extension to POSIX (Portable Operating System Interface) standard. The `#define _XOPEN_SOURCE 700` definition before the system header files specifies the usage of the XSI extension to the recent POSIX.1-2008 standard. The definition of the `_XOPEN_SOURCE` constant is required to use those mutex attributes.

[5] To use the error constants, the `errno.h` header file needs to be included. The error description can be obtained with the `strerror()` function, whose declaration is in `string.h`.

[6] The POSIX functions `pthread_mutexattr_setrobust()` and `pthread_mutexattr_consistent()` are not implemented in the GNU C Library for the Linux kernel (i.e. `glibc` package). The equivalent non-portable versions `pthread_mutexattr_setrobust_np()` and `pthread_mutexattr_consistent_np()` have to be used.

`pthread_mutex_consistent_np()`[6] marks the mutex state as consistent again. It has to be called before unlocking. If recovery is not possible and unlocking is done without `pthread_mutex_consistent_np()`[6], the next `pthread_mutex_lock()` fails with the `ENOTRECOVERABLE`[5] error.

```
err = pthread_mutexattr_setrobust_np(&attr, PTHREAD_MUTEX_ROBUST);
        ...
        ...
err = pthread_mutex_lock(&mutex);
if(err == EOWNERDEAD)
{
  process owning the mutex terminated, mutex is inconsistent, recover
          ...
  if(recovery successful) err = pthread_mutex_consistent_np(&mutex);
  else panic!
} else if(err == ENOTRECOVERABLE) panic!
        ...
err = pthread_mutex_unlock(&mutex);
```

The ownership is again the key for solving the priority-inversion problem (see subsection 7.1.4). The ownership alone is not enough, though. A priority-inversion prevention protocol must be used. The priority ceiling and priority inheritance are the most common priority-inversion avoidance protocols.

Priority ceiling

A priority is assigned to each mutex. The mutex priority is equal or higher than the priority of any process using the mutex. To prevent the priority-inversion, the process priority is boosted to the mutex priority when the process locks the mutex[7] and lowered back to its original value on the mutex unlock (Fig. 7.5). While holding the mutex, the process can be pre-empted only by a process not using the mutex (i.e. its priority is higher than the mutex's). Therefore, the priority inversion cannot happen.

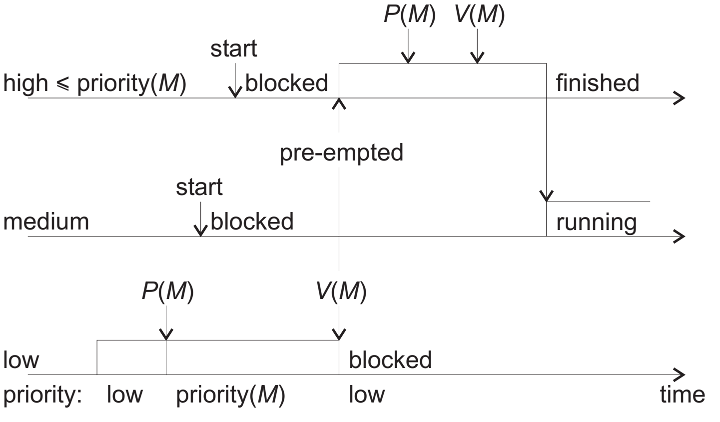

Figure 7.5: Priority ceiling protocol

The priority ceiling protocol solves the priority-inversion problem, but on the

[7]If a process locks more than one mutex with different priorities, then the process priority equals to the highest priority of the locked mutexes.

other hand causes a potential CPU time starvation. Example: The high- and low-priority processes use mutex M. The high-priority process locks mutex M rarely, while the low-priority process locks it frequently. Every time the low-priority process locks mutex M, its priority is boosted. In most cases, this would not be required since the high-priority process only rarely locks mutex M. Nevertheless, a low-priority process frequently blocks both, the high- and the medium-priority processes. After unlocking mutex M, the high-priority process immediately pre-empts the low-priority one, the medium-priority process has to wait further. Thus the high- and especially the medium-priority processes can miss their deadlines because of the unnecessary low-process priority boosts. The medium-priority process is most deprived.

The priority-ceiling protocol eliminates the circular deadlock (see subsection 7.1.3) since the processes holding mutex M run at the same priority. A circular dependence cannot develop although the mutexes are not ordered (see Fig. 7.6).

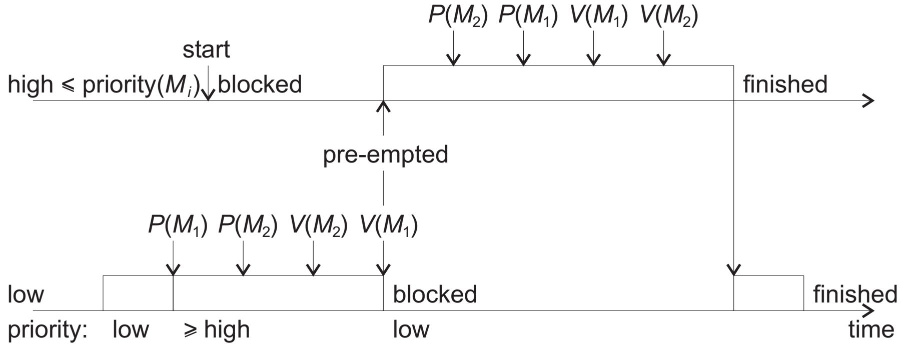

Figure 7.6: Circular dependence cannot develop with the priority-ceiling protocol

The priority-ceiling protocol is assigned to a mutex by the `PTHREAD_PRIO_PROTECT` attribute set with the `pthread_mutexattr_setprotocol()` function. The mutex priority is defined by the `pthread_mutexattr_setprioceiling()` function. If a mutex with the `PTHREAD_PRIO_PROTECT` attribute has a lower priority than the process trying to lock it, `pthread_mutex_lock()` returns the `EINVAL`[5] error.

```
err = pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_PROTECT);
err = pthread_mutexattr_setprioceiling(&attr, priority);
```

Only the processes with the real-time `SCHED_FIFO` and `SCHED_RR` scheduling policies (see page 117) can lock the mutex with the priority-ceiling protocol. This is because the Linux kernel boosts the process priority at the `pthread_mutex_lock()` call with an internal call to `sched_setscheduler()`. The latter returns the error if a non-zero priority is assigned to the non-real-time `SCHED_OTHER` scheduling policy.

Priority inheritance

The priority of a low-priority process is boosted only when a low-priority process holding the mutex in fact blocks the high-priority one waiting for it. A low-priority process priority is equaled to the priority of a blocked high-priority process. A low-priority process inherits a high-priority process priority[8]. Therefore, while blocking a high-priority process, a low-priority process cannot be blocked by a medium-priority one (see Fig. 7.7). A priority inversion cannot happen.

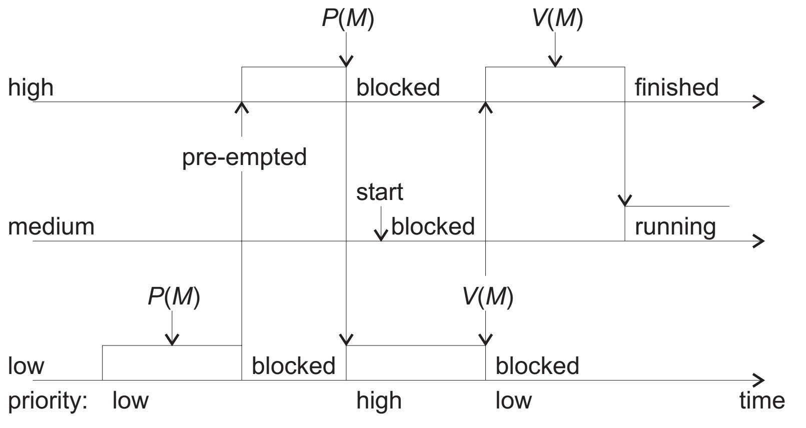

Figure 7.7: Priority-inheritance protocol

The priority-inheritance protocol does not cause the CPU time starvation. But it neither solves the circular deadlock like the priority-ceiling protocol. Nevertheless, the main drawback of the priority-inheritance protocol is the worst-case blocked time. Theoretically, it can rise to the sum of the mutex-protected critical sections of all lower-priority processes (see Fig. 7.8).

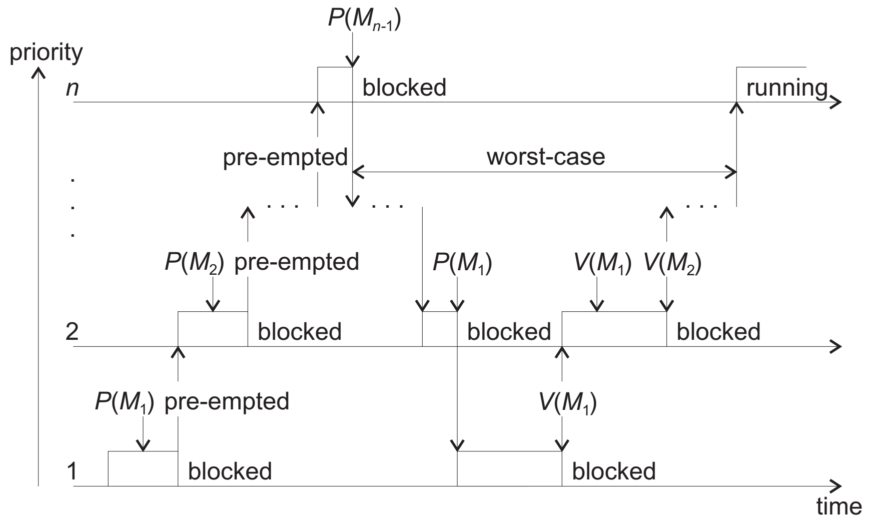

Figure 7.8: Worst-case blocked time with the priority inheritance

The priority-inheritance protocol is assigned to a mutex by the `PTHREAD_PRIO_INHERIT` attribute set with the `pthread_mutexattr_setprotocol()` function.

[8]The priority is inherited recursively. Example: The process with the lowest priority holds M_1. The high-priority process is blocked by a low-priority process because of M_2. At the moment, low-priority process is running with priority boosted to high, the lowest- and high-priority processes are blocked. If a low-priority process tries to lock M_1, it is blocked by the lowest-priority process. The priority of the lowest-priority process is not boosted only to low, but recursively to high.

```
err = pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
```

The additional mutex functionality defined with various attributes (error checking, recursion, robustness, priority ceiling and inheritance) is not priceless, though. With each attribute, locking and unlocking become slower.

Chapter 8

Loadable kernel modules

A loadable kernel module is a part of the kernel code which can be loaded and unloaded on demand. Support for a special hardware (i.e. device driver) or for the custom operating system service (i.e. custom system call) can be added by the kernel module. The loadable kernel modules extend the kernel functionality without rebooting.

The information about the currently loaded modules is available in the `/proc/modules` virtual file. The `lsmod` command prints `/proc/modules` in a readable format. A single module is installed into the kernel by the `insmod` command, e.g. the `dummy.ko` module with the command-line `name` parameter is installed with:

```
insmod ./dummy.ko name="circbuf"
```

The module is removed from the kernel with the `rmmod` command, e.g:

```
rmmod dummy
```

The high-level loadable kernel module handling `modprobe` command can be used instead of `insmod` and `rmmod`. `modprobe` checks the configuration file to find an appropriate module(s) from a specified generic identifier. It also uses dependency file to load the dependent modules that must be loaded before the requested module. `modprobe` can remove the unused auto-loaded modules, etc. The loadable kernel module handling `insmod`, `rmmod` and `modprobe` commands can be performed only by a super user.

8.1 Kernel module programming

A detailed explanation of writing the kernel module code [54, 55] far exceeds the scope of this textbook. An explanation of a simple circular buffer device driver code follows to taste the matter. The module creates a dummy circular buffer device in the `/dev` directory.

The kernel module must have the `init_module()` and `cleanup_module()` functions. The former performs module initialization and is called at the module installation into the kernel. The latter is called at the module removal from the kernel to undo the module specifics. `init_module()` must return zero on success. A nonzero value indicates an error and the module is therefore not installed.

```
/* dummy.c */
#include "dummy.h"
#include <linux/fs.h>
```

```
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <linux/miscdevice.h>

static int begin = 0, end = 0, size = 0;
static char *buffer = NULL, *name = "dummy";
module_param(name, charp, 0);

         ...

static struct file_operations fops = {
  .owner = THIS_MODULE,
  .open = dummy_open,
  .ioctl = dummy_ioctl,
  .read = dummy_read,
  .write = dummy_write,
  .release = dummy_release
};
static struct miscdevice dev = {MISC_DYNAMIC_MINOR, NULL, &fops};

int init_module()
{
  dev.name = name;
  return misc_register(&dev);
}

void cleanup_module()
{
  kfree(buffer);
  misc_deregister(&dev);
}

MODULE_LICENSE("GPL");
```

A command-line argument (see the `insmod` example on page 155) is passed to the kernel module global variables by the `module_param()` macro. The macro takes three parameters: global variable, its type (e.g. `charp` for type `char *`) and access rights to the `/sys/modules/`*module_name*`/parameters/`*parameter_name* virtual file with parameter value. If the access rights are not given, the parameter is not exported and a virtual file is not created. In the code above, the `name` command-line argument is passed to the `name` global pointer.

In Linux, a device is represented with a file by a convention located in the `/dev` directory. The device is identified by a major and minor number listed by the `ls -l` command. The device driver handling the device registers has a unique major number. The major number identifies the device driver. The minor numbers are used for the devices handled by the same driver (e.g. `/dev/sda1` and `/dev/sda2` hard disk partitions have the same major and different minor numbers, since they are two devices handled by the same driver). An exception is a miscellaneous device driver with the major number `10`. It is a set of small character device drivers each handling one minor number.

The loadable kernel module is an object[1] file (see page 34) linking into the ker-

[1]By convention, the kernel modules have the `.ko` (i.e. kernel object) extension to distinguish from the conventional object files with the `.o` extension.

nel with insmod. Thus, the symbols[2] used in the code are resolved upon installing the module. This further means that only the external symbols defined in the kernel can be used. The standard C library functions cannot be used. Another consequence of linking against the entire kernel is a uniform name space. The global symbol defined in the loadable kernel module code is seen to the entire kernel[3] and must therefore be unique. Naming such symbols with a unique module-defined prefix[4] is recommended. To keep the global symbol private to the module, declare it as static. The list of the available symbols is in /proc/kallsyms. The loadable kernel module shares the kernel code space[5]. It does not have its own memory. Thus, the module segmentation fault is the kernel segmentation fault. An uncontrolled memory writing can easily corrupt the kernel data. Therefore, the loadable kernel modules should be coded with an extreme caution.

In the code on page 155, a small device driver is registered with a miscellaneous driver by the kernel-provided misc_register() function at module initialization. A unique minor number is automatically selected (MISC_DYNAMIC_MINOR). The driver is unregistered by misc_deregister() at the module removal.

A miscellaneous device is represented by the miscdevice structure. The second element is the device name (set in init_module()), the third is the file_operations structure. The latter is a collection of the function pointers. A particular function is called on the corresponding system call (e.g. open(), read(), etc.) to the device. The C language standard version C99 enables a more readable defining of the structure elements[6]. The only element in the file_operations structure not being a function pointer is the owner element. It points to the module that owns the operations. It is used by the kernel to prevent the module removal while its operations are in use.

The Linux kernel is released under GNU GPL (GNU General Public License). This means that the source code is available and can be modified in any way. The modifications, however, can be distributed further only under GNU GPL. A loadable kernel module is actually a part of the kernel and can be compiled into it. If a kernel extended with a module is distributed, it must be under GNU GPL. Thus, the proprietary modules should not be distributed as a part of the kernel. The MODULE_LICENSE() macro defines the module license. When a proprietary module is loaded into the kernel, a tainted kernel message is issued to warn the user that the loaded software is not free. If MODULE_LICENSE() is not specified, the module is considered proprietary.

The code handling system calls to a simple circular buffer device is needed to finish an example device driver.

```
        ...

static int dummy_open(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "/dev/%s opened by pid %d\n", name, current->pid);
  return 0;
}
```

---

[2] The symbol is a variable or a function.

[3] The global symbol can be used by the kernel code, but not by other loadable kernel modules. To make it visible to other modules, the symbol has to be exported by the EXPORT_SYMBOL() macro. All said about the global symbols is true for the functions by default. The kernel must be configured with CONFIG_KALLSYMS_ALL to support the global variables.

[4] The kernel prefixes are a lowercase by convention.

[5] This is not true for the microkernels (see page 77).

[6] E.g., the .open = dummy_open line defines the open structure element which is a pointer to the function that is called on the open() system call to the device. Thus, the dummy_open() module function is called on the open() system call.

```
static int dummy_ioctl(struct inode *inode, struct file *file,
                                unsigned int cmd, unsigned long arg)
{
  if(cmd == GET_SIZE) *((int *)arg) = size;
  if(cmd == SET_SIZE)
  {
  begin = 0, end = 0, size = 0;
    buffer = krealloc(buffer, arg, GFP_KERNEL);
    if(buffer == NULL) return -ENOMEM;
    size = arg;
  }
  if(cmd == NUM)
  {
    int num = end - begin;
    if(num < 0) num = num + size;
    *((int *)arg) = num;
  }
  return 0;
}

static ssize_t dummy_read(struct file *file, char *buf,
                                         size_t count, loff_t *ppos)
{
  int i;
  for(i = 0; i < count && begin != end; i = i + 1)
  {
    put_user(buffer[begin], buf + i);
    begin = begin + 1;
    if(begin >= size) begin = 0;
  }
  return i;
}

static ssize_t dummy_write(struct file *file, const char *buf,
                                         size_t count, loff_t *ppos)
{
  int i, tmp;
  for(i = 0; i < count; i = i + 1, end = tmp)
  {
    tmp = end + 1;
    if(tmp >= size) tmp = 0;
    if(tmp == begin) break;
    get_user(buffer[end], buf + i);
  }
  return i;
}

static int dummy_release(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "/dev/%s closed by pid %d\n", name, current->pid);
  return 0;
}

         ...
```

The `dummy_open()` and `dummy_release()` functions are called on the `open()` and `close()` system calls, respectively. Both functions issue[7] a message with the `prink()` kernel function. The `current` kernel macro provides a pointer to the structure with the calling process data (e.g. PID, name, process state, etc.).

The `dummy_read()` and `dummy_write()` functions are called on the `read()` and `write()` system calls. They implement a circular FIFO buffer. The data is stored in the `buffer` with a space for `size` characters. The `begin` and `end` indices point to the oldest character and the first empty space, respectively. Both functions return the number of characters read or written. The `put_user()` and `get_user()` kernel functions are used to copy the buffer data from the kernel address space to the process address space and vice versa. The ordinary assignment (i.e. `buf[i] = buffer[begin];`) would not do.

The most interesting is the `dummy_ioctl()` function called on the `ioctl()` system call. It handles the device properties defined in `dummy.h` with the `_IOR()` macro.

```
/* dummy.h */
#include <linux/ioctl.h>
#define GET_SIZE _IOR(0, 0, int *)
#define SET_SIZE _IOR(0, 1, int)
#define NUM      _IOR(0, 2, int *)
```

The `GET_SIZE` property retrieves the current buffer size. `SET_SIZE` resets the buffer and sets a new buffer size. A corresponding memory space is allocated by `krealloc()`[8]. `dummy_ioctl()` returns `-ENOMEM` if allocation fails. In this case, `ioctl()` returns `-1` with the `errno` set to `ENOMEM`. The allocated memory is freed with `kfree()`[8] on the module removal in `cleanup_module()`. The `NUM` property retrieves the number of characters currently stored in the buffer.

With the described module installed into the kernel, the simple circular buffer device can be used by a process as any other device/file.

```
#include "dummy.h"
        ...
char buf[SIZE];
int dev = open("/dev/circbuf", O_RDWR);
int err = ioctl(dev, SET_SIZE, 10);
        ...
int n = write(dev, buf, 5);
        ...
err = ioctl(dev, NUM, &n);
n = read(dev, buf, n);
        ...
err = close(dev);
```

8.2 Compiling a kernel module

A kernel module is compiled for a particular kernel version. If the module is compiled for a running system, then the running kernel header files are needed.

[7]The message is displayed on the console if the loglevel (i.e. `KERN_INFO`) is below a certain value. If the system and kernel `syslogd` and `klogd` logging daemons are running, the message is also written into `/var/log/messages`.

[8]Kernel versions of `realloc()` and `free()`.

They can be found in the `/lib/modules/`*kernel_version*`/build`[9] directory. The following `Makefile` is needed in the working directory with the module source file.

```
# Makefile
KSRC=/lib/modules/kernel_version/build
obj-m := dummy.o
all:
	make -C $(KSRC) M=$(PWD) modules
clean:
	make -C $(KSRC) M=$(PWD) clean
```

The `Makefile` above suits a module with one source file (i.e. `dummy.c`). If the kernel module source code consists of more than one file, they have to be stated.

```
         ...
obj-m := module_name.o
module_name-objs := scr1.o src2.o ...
         ...
```

With the `Makefile` ready, a cleanup and building loadable kernel module object file from a scratch is performed by two `make` commands.

```
make clean
make
```

8.2.1 Cross-compiling a kernel module

A kernel module can be cross-compiled for the target system on a host system (see section 4.3). The cross compiler, target architecture and directory with the target kernel source have to be specified in `Makefile`. To cross-compile for the Phytec phyCORE-i.MX27 development kit, `Makefile` has to be modified.

```
# Makefile (cross compile for phyCORE-i.MX27)
KSRC=...OSELAS.BSP-Phytec-phyCORE-i.MX27-PD11.1.1/
                 platform-phyCORE-i.MX27/build-target/linux-2.6.38[10]
obj-m := dummy.o
all:
	make -C $(KSRC) ARCH=arm
	       CROSS_COMPILE=arm-v5te-linux-gnueabi- M=$(PWD) modules
clean:
	make -C $(KSRC) M=$(PWD) clean
```

The command-line `ARCH` and `CROSS_COMPILE` arguments are added. They specify the ARM architecture[11] and the corresponding cross-compiler prefix. The path to the cross-compiler executable (i.e. `arm-v5te-linux-gnueabi-gcc`) has to be included in the command search path[12].

[9] If the running kernel header files are not installed, install the `linux-headers-`*kernel_version* package (see section 1.11). The *kernel_version* can be obtained by the `uname -r` command.

[10] Subdirectory with the target kernel source on the host.

[11] The Phytec phyCORE-i.MX27 development kit is based on the ARM926EJ-S architecture.

[12] The `arm-v5te-linux-gnueabi-gcc` cross compiler resides in the `/opt/OSELAS.Toolchain-2011.02.0/arm-v5te-linux-gnueabi/gcc-4.5.2-glibc-2.13-binutils-2.21-kernel-2.6.36-sanitized/bin` directory which should be included in the `PATH` variable.

Bibliography

[1] M. Garrels, *Introduction to Linux*, 1.27 ed., 2008, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[2] A. Weeks, *The Linux System Administrator's Guide*, Ver. 0.9, 2004, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[3] D.M. Ritchie, K. Tompson, *The UNIX Time-Sharing System*, CACM, Vol. 17 No. 7, pp. 365-375, 1974

[4] T. Aivazian, *Linux Kernel 2.4 Internals*, 2002, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[5] S.R. Bourne, *An Introduction to the UNIX Shell*, BSTJ, Vol. 57, No. 6, Part 2, pp. 2797-2822, 1978

[6] *Filesystem Hierarchy Standard*, ed. R. Russel, D. Quinlan, C. Yeoh, 2004, *http://www.pathname.com/fhs/pub/fhs-2.3.pdf*, Jul. 2014

[7] B. Nguyen, *Linux Filesystem Hierarchy*, Ver. 0.65, 2004, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[8] V. Prabhakaran, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, *Analysis and Evolution of Journaling File Systems*, Proc. USENIX 2005 Annual Technical Conf., pp. 105-120, 2005

[9] G. Anderson, *GNU/Linux Command-Line Tools Summary*, Rev. 1.2, 2006, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[10] The Linux man-pages project *https://www.kernel.org/doc/man-pages/*, Jul. 2014

[11] Linux manpages *http://www.linuxmanpages.com/*, Jul. 2014

[12] W. Joy, M. Horton, *An Introduction to Display Editing with Vi*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1980

[13] M. Garrels, *Bash Guide for Beginners*, Ver. 1.11, 2008, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[14] M. Cooper, *Advanced Bash-Scripting Guide*, Rev. 10, 2014, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[15] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, 2^{nd} ed., Prentice Hall, 1988

[16] B. Stroustrup, *The C++ Programming Language*, 3^{rd} ed., Addison-Wesley, 1997

[17] R.M. Stallman, *Using the GNU Compiler Collection*, GNU Press, 2014, Free Software Foundation, *https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc.pdf*, Jul. 2014

[18] R. Stallman, R. Pesch, S. Shebs, *Debugging with GDB*, Free Software Foundation, 2014, Free Software Foundation, *http://sourceware.org/gdb/download/onlinedocs/gdb.pdf.gz*, Jul. 2014

[19] Debian, The universal operating system, *https://www.debian.org/*, Jul. 2014

[20] O. Kirch, T. Dawson, *Linux Network Administrators Guide*, 2000, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[21] *IEEE Standard for Ethernet 802.3^{TM}-2012*, IEEE Computer Society, 2012

[22] RFC 1122, *Requirements for Internet Hosts - Communication Layers*, IETF, Network Working Group, ed. R. Braden, 1989, *http://www.rfc-editor.org/pdf rfc/rfc1122.txt.pdf*, Jul. 2014

[23] RFC 1123, *Requirements for Internet Hosts - Application and Support*, IETF, Network Working Group, ed. R. Braden, 1989, *http://www.rfc-editor.org/pdfrfc/rfc1123.txt.pdf*, Jul. 2014

[24] B.A. Forouzan, *TCP/IP Protocol Suite*, 4$^{\text{th}}$ ed., McGraw-Hill, 2010

[25] IANA, Service Name and Transport Protocol Port Number Registry, *http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml*, Jul. 2014

[26] RFC 1918, *Address Allocation for Private Internets*, IETF, Network Working Group, Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, 1996, *http://www.rfc-editor.org/pdfrfc/rfc1918.txt.pdf*, Jul. 2014

[27] RFC 919, *Broadcasting Internet Datagrams*, IETF, Network Working Group, J. Mogul, 1984, *http://www.rfc-editor.org/pdfrfc/rfc919.txt.pdf*, Jul. 2014

[28] RFC 1034, *Domain Names - Concepts and Facilities*, IETF, Network Working Group, P. Mockapetris, 1987, *http://www.rfc-editor.org/pdfrfc/rfc1034.txt.pdf*, Jul. 2014

[29] RFC 1035, *Domain Names - Implementation and Specification*, IETF, Network Working Group, P. Mockapetris, 1987, *http://www.rfc-editor.org/pdfrfc /rfc1035.txt.pdf*, Jul. 2014

[30] RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*, IETF, Network Working Group, T. Berners-Lee, R. Fielding, L. Masinter, 1987, *http://www.rfc-editor.org/pdfrfc/rfc3986.txt.pdf*, Jul. 2014

[31] RFC 6761, *Special-Use Domain Names*, IETF, S. Cheshire, M. Krochmal, 2013, *http://www.rfc-editor.org/pdfrfc/rfc6761.txt.pdf*, Jul. 2014

[32] RFC 2131, *Dynamic Host Configuration Protocol*, IETF, Network Working Group, R. Droms, 1997, *http://www.rfc-editor.org/pdfrfc/rfc2131.txt.pdf*, Jul. 2014

[33] D.J. Barrett, R.E. Silverman, *SSH, the Secure Shell: The Definitive Guide*, O'Reilly, 2001

[34] B. Callaghan, *NFS Illustrated*, Addison-Wesley, 2000

[35] World Wide Web Consortium (W3C), *http://www.w3.org/*, Jul. 2014

[36] N.C. Zakas, *Professional JavaScript® for Web Developers*, 3rd ed., Wiley, 2012

[37] Official PHP Website, *http://php.net/*, Jul. 2014

[38] O. Andreasson, *Iptables Tutorial 1.2.2*, 2006, Frozentux, *https://www.frozentux.net/iptables-tutorial/iptables-tutorial.ps.gz*, Jul. 2014

[39] X.Org Foundation, *http://www.x.org/wiki/*, Jul. 2014

[40] A. Coopersmith, M. Dew, *The X New Developer's Guide*, ed. B. Massey, X.Org Foundation, 2012, *http://www.x.org/wiki/guide/guide.epub*, Jul. 2014

[41] phyCORE®-i.MX27, PHYTEC Messtechnik GmbH, *http://www.phytec.com/products/system-on-modules/phycore/i.mx27/*, Jul. 2014

[42] phyCORE-i.MX27 Linux Board Support Package, *ftp://ftp.phytec.de/pub/Products/phyCORE-iMX27/Linux/*, Jul. 2014

[43] The Barebox Bootloader, *http://barebox.org/*, Jul. 2014

[44] D. Woodhouse, *JFFS: The Journaling Flash File System*, Proc. OLS, 2001

[45] Advanced Linux Sound Architecture project, *http://www.alsa-project.org/*, Jul. 2014

[46] GStreamer open source multimedia framework, *http://gstreamer.freedesktop.org/*, Jul. 2014

[47] J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4*, Prentice Hall, 2006

[48] J.W.S. Liu, *Real-Time Systems*, Prentice-Hall, 2000

[49] Real-Time Linux Wiki, *https://rt.wiki.kernel.org/index.php/Main_Page*, Jul. 2014

[50] B. Hall, *Beej's Guide to Unix IPC*, 2010, *http://beej.us/guide/bgipc/output/print/bgipc_A4_2.pdf*, Jul. 2014

[51] *Standard for Information Technology - Portable Operating System Interface (POSIX®)*, IEEE and The Open Group, 2008

[52] E.W. Dijkstra, *Cooperating Sequential Processes*, Technological University, Eindhoven, The Netherlands, 1965, reprinted in *Programming Languages*, ed. F. Genuys, str. 43-112, Academic Press, 1968

[53] E.W. Dijkstra, *Solution of a Problem in Concurrent Programming Control*, CACM, Vol. 8, No. 7, p. 569, 1965

[54] P.J. Salzman, O. Pomerantz, *The Linux Kernel Module Programming Guide*, Ver. 2.4.0, for kernel ver. 2.4, 2003, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

[55] P.J. Salzman, M. Burian, O. Pomerantz, *The Linux Kernel Module Programming Guide*, Ver. 2.6.4, for kernel ver. 2.6, 2007, The Linux Documentation Project, *http://www.tldp.org/guides.html*, Jul. 2014

Janez Puhan *is a teaching assistant at the Faculty of Electrical Engineering, University of Ljubljana.*

- introduction into general principles of operating systems and network configuration
- Linux operating system used as an example platform
- PHYTEC phyCORE-i.MX27 development kit used as an embedded system platform
- explanation of real-time concepts with real-time application development
- interprocess communication and simultaneous resource access with presentation of deadlock situations
- providing a flying start to a beginner using the POSIX-compliant C code examples

ISBN 978-961-243-275-1

9 789612 432751