Unit 1

Quantitative analysis of computer performance

Labs of
Computer Architecture
v1.2

SESSION 1

Introduction to the work environment

Objectives

This lab session shows the student the development environment that will be used in most of the labs of the course. This environment is based on a GNU/Linux operating system and the C programming language. Therefore, the main goals of this session are:

- Introducing the GNU/Linux operating system and its basic commands.
- Introducing the basic concepts of the C programming language.
- Understanding the operation of pointers and their arithmetic in C.

In order to achieve these goals a Linux operating system will be launched. Then, several small programs will be written in C.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.
- Attend the lab session with a USB memory stick to store the files created while working on it.

Session development

1. Loading the operating system

GNU/Linux is a multitasking and multiuser operating system. This means that it is able to execute several tasks owned by different users at the same time. It is a Unix-like operating system, so it shares lots of features with other operating systems such as BSD or Mac OS X.

The operating system can be loaded from a previous installation in a hard disk, or using a *Live-CD/DVD* which allows running the operating system without installation. The latter will be the option chosen in this course.

❏ Insert the course DVD in the optical drive of your computer.

❏ Reboot the computer (a soft or warm reboot is preferred; please do not trigger a hardware-based reset) or turn it on.

❏ Select the optical drive as the computer boot device. Some computers show an option menu while booting by pressing F12. If this menu does not appear, the preference of the optical drive in the boot sequence of the BIOS should be higher than the preference of other devices.

While booting from the *Live-DVD*, a menu showing two options is displayed. In the computer of the laboratory the option to be chosen is `Probar Ubuntu` (`Test Ubuntu`), since the operating system will be run without modifying the computer configuration. The other option, `Instalar Ubuntu` (`Install Ubuntu`), starts the installation process of the operating system in the computer (you can use this option if you want to install the operating system in your personal computer).

❏ Select the option `Probar Ubuntu`.

GNU/Linux provides the user with two working modes: graphical mode and text mode. In the graphical user interface, or GUI, you can interact with the system using the desktop, like in other windows-based operating systems. Usually, the graphical mode is used to work in local mode, that is, when staying in front of the computer. The text mode provides the same functionality as the graphical mode but the interaction with the system is carried out through a text-based command line interface, or CLI. This mode is commonly used when working with the system remotely, or when working with servers in order to save system resources.

Although it seems that GNU/Linux provides a unique user interface, graphic- or text-based, the operating system provides several interfaces by default. These interfaces are called consoles, numbered from 1 onward. Typically, console #1 is a text-mode console whereas console #7 is the graphical console showing the desktop. You can show different consoles by pressing CRTL + ALT + function key.

- ❑ Switch to console #1 in text mode by pressing CRTL + ALT + F1.
- ❑ Return to the graphical mode choosing the console #7 by pressing CRTL + ALT + F7.

You can switch between the local consoles using the function keys as shown above. Consoles are also created when connected to remote systems, by means of the `SSH` protocol for example.

Given the flexibility provided by the command line interface of the operating system, and taking into account that the graphical interface will not be available in all systems, the basic commands of the command line interface will be described next.

2. Command line interface

When the system is accessed in text mode, the command line interface of the operating system appears. It is also possible to launch a command line interface in the graphical mode, although the process can vary depending on the desktop environment chosen.[1] The GNU/Linux used in this course is an Ubuntu distribution using GNOME; the option `Terminal` in `Aplicaciones`→`Accesorios` (`Applications`→`Accessories`) is used to launch the command line interface. Figure 1.1 shows this step.

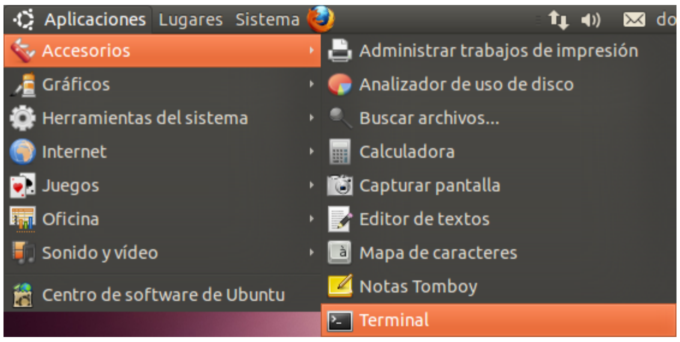


Figura 1.1: Opening a command line interface in the graphical mode

UNIX allows using different command line interfaces, although the most commonly used is the *bash* interface. This has several implications in the development of *batch* tasks, that is, while programming the command line interface through *scripts*[2].

2.1. Working folder

Once the command line interface is open, the *prompt* is shown. The prompt is the text displayed by default at the beginning of the line showing the current

[1] GNU/Linux allows using several desktop environments, such as GNOME, KDE or Unity.
[2] A *script* is a text file containing commands.

working directory. All users have a working directory assigned by default. It is called `home` and it is represented in the prompt with the character '~'. The `pwd`[3] command shows the current working directory:

```
$> pwd
/home/ubuntu
```

The concept of working directory is vital, since all file access operations (creating or deleting a file, for example) will be carried out over this directory by default.

As it happens in other file systems, in UNIX the files are grouped in directories hierarchically organized.[4] Unlike in Windows operating systems, in UNIX neither the physical drives nor the hard disk partitions are labeled as `C:` or `D:`, but all directories hang from the root directory, represented as `/`. To change the working directory the `cd` command is used.

- ❏ Change the working directory to the root directory

  ```
  $> cd /
  ```

- ❏ Go down one level in the directory hierarchy to `/etc`

  ```
  $> cd etc
  ```

- ❏ Go to the directory `/usr/include` specifying its absolute path

  ```
  $> cd /usr/include
  ```

- ❏ Go back to the previous directory

  ```
  $> cd -
  ```

- ❏ Go up one level in the directory hierarchy. Be careful, there is a blank between the name of the command and the two points

  ```
  $> cd ..
  $> pwd
  /
  ```

- ❏ Go to the directory `home`, that is, the default working directory when you launch the command line interface. This can be done in several ways:

  ```
  $> cd
  $> cd ~
  $> cd $HOME
  ```

There are two ways of specifying a path in the directory hierarchy: absolute and relative. It is possible to specify an absolute path writing the set of directories that is necessary to pass through to reach the destination file from the the root directory. For instance, the following example uses the `cat` command, that shows the content of a file, specifying its absolute path. Therefore, this command will operate properly regardless the current working directory of the user.

[3]Note that the command line interface is case-sensitive.
[4]In Windows the directories are usually called folders.

```
$> cat /etc/hostname
ubuntu
```

On the other hand, the special directories `.` and `..` can be used to specify relative paths. The former represents the current directory whereas the latter represents the parent directory. A relative path does never begin with the root directory, `/`, as it happens in an absolute path.

```
$> cd
$> pwd
/home/ubuntu
$> cat ../../etc/hostname
ubuntu
$> cd /etc
$> cat hostname
ubuntu
$> cd ..
$> cat ./etc/hostname
ubuntu
$> cat etc/hostname
ubuntu
```

The contents of a directory can be listed by means of the `ls` command.

```
$> ls
bin   cdrom  etc   initrd.img  media  opt   rofs  sbin     srv  tmp  var
boot  dev    home  lib         mnt    proc  root  selinux  sys  usr  vmlinuz
```

Most of the UNIX systems organize their files in a similar way, showing directories hanging from the root directory with the following purpose:

- **`/home`**. Directory where user directories are stored. Usually, one directory per user is created.
- **`/bin`**. Contains commands that can be executed, like the `pwd` command.
- **`/dev`**. Directory where special files used to access the drives or peripherals of the computer are stored.
- **`/etc`**. Contains most of the configuration files of the operating system as well as the installed programs.
- **`/proc`**. Files with information about the system.

2.2. Editing files

Editing a text file can be done by means of `gedit` in the graphical mode or by means of the `nano` editor in the text mode.

A file will be edited using the editor in the text mode. Be aware of the current working directory, since it is where the file is created.

- ❏ Open the editor with the name of the new file, `1-1text.txt`. Use the option `-c` for the editor to show the line numbers of the file at the bottom.

  ```
  $> cd
  $> nano -c 1-1texto.txt
  ```

- ❏ Write any sentence in the file.
- ❏ Save the file by pressing CRTL + O and then press Enter to confirm the name of the file.[5]
- ❏ Close the editor by pressing CRTL + X.

As with any other command, an absolute or a relative path can be used to specify the location of the file. In the previous example, a relative path was used. Next, the same command is executed using the absolute path of the file.

- ❏ Open the file created in the previous example using an absolute path.

  ```
  $> nano -c /home/ubuntu/1-1text.txt
  ```

- ❏ Close the editor.

2.3. Permissions

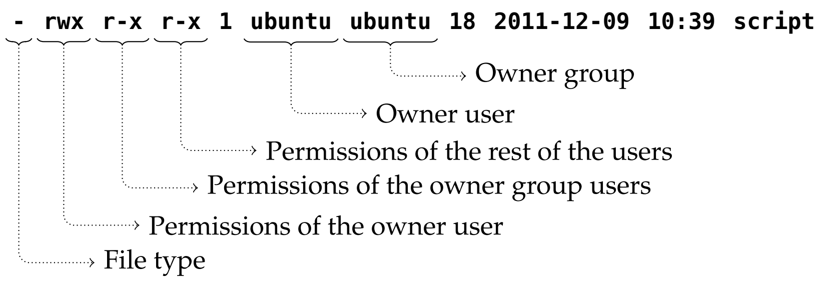
As other multiuser operating systems, UNIX labels each input in the file system (file, directory of symbolic link) with a sequence of permissions. There are three types of permissions: read permission (`r`), write permission (`w`), and execute permission (`x`). Furthermore, these permissions are assigned in three different levels: the file (or directory or symbolic link) owner, the owner group, and the rest of the users of the system.

The `ls` command shows the attributes of the files (permissions, owner, etc.) when invoqued with the option `-l`.

```
$> ls -l
total 12
drwxr-xr-x 2 ubuntu ubuntu 40 2012-12-09 10:42 directory
-r--r--r-- 1 ubuntu ubuntu 43 2012-12-09 10:07 1-1text.txt
-rw-r--r-- 1 ubuntu root   16 2012-12-09 10:40 other.txt
-rwxr-xr-x 1 ubuntu ubuntu 18 2012-12-09 10:39 script
```

The information shown in the above list is organized in columns as follows:

[5]The editor also shows at the bottom the shortcuts for the most commonly used operations; the character ^ represents the key Control.



```
- rwx r-x r-x 1 ubuntu ubuntu 18 2011-12-09 10:39 script
```

Owner group
Owner user
Permissions of the rest of the users
Permissions of the owner group users
Permissions of the owner user
File type

The first element on the previous list obtained by `ls` is a directory, and thus it is identified by the character `d` at the beginning of its row; the rest of the elements are regular files, identified by the character `-`. The symbolic links are identified by the character `l`.

The `ubuntu` user is the owner of the four elements of the list. Furthermore, this user has read and write permissions over all the files except for `1-1text.txt` and for the directory. In addition, the user has execution permission over the directory and the file `script`.

The permissions of an element can be changed by means of the `chmod` command. For instance, write permissions can be given to `1-1text.txt` as follows:

```
$> chmod +w 1-1text.txt
$> ls -l 1-1text.txt
-rw-r--r-- 1 ubuntu ubuntu 43 2011-12-09 10:07 1-1text.txt
```

The permissions can also be modified considering each triplet `xyz` as a binary number of 3 bits, representing whether each permission on the triplet is allowed, as shown in the next example:

- Owner user: all permissions $\Rightarrow$ `rwx` $\overset{\text{binary}}{\Rightarrow}$ `111` $\overset{\text{decimal}}{\Rightarrow}$ `7`.
- Owner group: read and write $\Rightarrow$ `rw-` $\Rightarrow$ `110` $\Rightarrow$ `6`.
- Rest of users: read only $\Rightarrow$ `r--` $\Rightarrow$ `100` $\Rightarrow$ `4`.

```
$> chmod 764 1-1text.txt
$> ls -l 1-1text.txt
-rwxrw-r-- 1 ubuntu root   16 2011-12-09 10:40 1-1text.txt
```

Usually, launching a task or executing a command requires certain privileges that a common user does not have. Thus, a SuperUser, like the Administrators group in a Windows operating system, is required. In UNIX-like systems this user is `root`. However, the log in process is rarely done as `root`.

❏ Check the user that has logged in the system. Use the `whoami` command.

```
$> whoami
ubuntu
```

UNIX-like systems provide the ability to execute any command or program with `root` privileges by means of the `sudo` command. For instance, rebooting the system network logic can be done by executing the script `/etc/init.d/networking`. Although this script can be executed by all the users of the system, its inner commands require `root` privileges. Next, the system network logic will be stopped and rebooted.

❏ Try to reboot the system network logic by means of invoking the script `/etc/init.d/networking`. An error occurs since the user does not have the required privileges.

```
$> /etc/init.d/networking restart
```

❏ Now, invoke the script with SuperUser privileges.

```
$> sudo /etc/init.d/networking restart
```

The `sudo` command will be frequently used in the course labs.

2.4. Basic operations with files

The process of creating and editing text files, listing contents of directories and changing the current working directory has been described above. Some basic operations with files, such as copying, moving, renaming and deleting files will be described next.

❏ Create a directory named `dir` in the user working directory. The `mkdir` command is required. Like the rest of the commands, an absolute or a relative path can be used.

```
$> mkdir ~/dir
```

❏ Make a copy of the file `1-1text.txt` in the created directory named `1-1copy1.txt`. The `cp` command is required to do this. Analyze the following command invocation and try to understand all of its parameters. If you have any doubt, ask your teacher for help.

```
$> cd ~/dir
$> cp ../1-1text.txt ./1-1copy1.txt
```

❏ Rename the file `1-1copy1.txt` to `1-1copy2.txt` at the same time that it is moved to your `home` directory. The `mv` command is required. This command can be used for either moving or renaming files.

```
$> mv 1-1copy1.txt ../1-1copy2.txt
```

❏ Delete the file `1-1copy2.txt` by means of the `rm` command. Be careful with this command since it does not require confirmation when deleting files.

```
$> cd ..
$> rm 1-1copy2.txt
```

❏ Delete the `dir` directory. This command requires the directory to be empty.

```
$> rmdir dir
```

2.5. On-line help

GNU/Linux is an operating system oriented to C developers. In fact, the distributions of the system usually include a C compiler and development environment, together with other programming resources. One of these resources is the on-line help, which covers C programming issues as well as commands of the command line interface.

The on-line help of UNIX-like systems is organized in categorized manuals. Each category is numbered from 1 to 8. For example, category 1 covers commands of the command line interface whereas category 3 covers library functions callable from a C program.

You can show the on-line help for any command by means of the `man` command followed by the name of the command.

❏ Show the on-line help for the `ls` command. You can exit the manual by pressing `Q`.

```
$> man ls
```

❏ You can also provide the category to search in. Query the manual for the `printf` function in C. This function outputs a formated text.

```
$> man 3 printf
```

The manual shows useful information about how to call the function as well as the header files that must be used to properly build the program.

❏ Close the manual.

The category must be provided explicitly to distinguish between commands and C functions with the same name. In the example above, the category points to the C function.

in the example above to distinguish between the command and the

These help resources can also be queried through Internet search engines.

3. Programming in C

The programming language used in this course is C. C is by far the most used language in system programming. In this section some basic C concepts will be studied.

3.1. Hello, World!

You are about to write and run what it is called the first program in C. As the tradition dictates, it must be a "Hello World" program.

❑ Edit a file named `1-1hello.c` with the following content:

```
#include <stdio.h>

int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

You need to compile the source file and link the object file to generate the executable file of the program.

- Compiling: the source code written in the programming language (C in this case) is transformed into another computer language (the target language), in this case object code (machine code). Compiling a C source file requires invoking `gcc` compiler with `-c` flag:

```
$> gcc -c 1-1hello.c
$> ls *.o
1-1hello.o
```

- Linking: the object files are combined into an executable file. Linking the files requires invoking `gcc` again with the object files as parameters in the command line. The `-o` flag allows providing a name for the executable file. Otherwise, the executable file will be named `a.out`. The libraries used in the linking process are also indicated in this step. However, in this example no additional libraries are used. The object files are linked with the C standard library by default, which contains the object code of C standard functions such as `printf`.

```
$> gcc 1-1hello.o -o 1-1hello
$> ls -l 1-1hello
-rwxr-xr-x 1 ubuntu ubuntu 7155 2012-09-16 16:26 1-1hello
```

As you can see, the executable file called `1-1hello` is generated. Since the working directory is not included in the directory list used by the command line interface to look for programs to execute, you must indicate the path to the executable file in the command line interface.

❑ Try to run the program by specifying only its name. An error will occur:

```
$> 1-1hello
1-1hello: command not found
```

❑ Run the program specifying the path of the executable file, that is, the current directory. By doing so, the command line interface knows where to look for the file.

```
$> ./1-1hello
Hello, World!
```

❑ Delete the following files: `1-1hello.o` and `1-1hello`. Use the `rm` command.

You must remember always that you need to specify the path for the file for running your programs.

3.2. Calling library functions

Developing programs in C, as in almost any other programming language, requires calling functions that are not implemented in the program itself. This is the case of functions used to output characters to the screen, dynamically allocate memory, manage strings, etc. It makes no sense that the programmer implements these commonly used functions in each new program. The programming language defines a mechanism to reuse these functions.

These commonly used functions are implemented in separated files that are linked with the object file of the program during the linking phase. These files may be object files or library files (several objects files joined in one). During the compilation phase, the compiler needs to know the prototype of these functions used but not defined by the programmer. To do so, header files are used. These files contain the declaration of the functions, that is, their prototypes. This gives the compiler information regarding the number and types of the parameters of the function.

In the linking process the linker needs to know what object files of libraries are required by these functions. Fig. 1.2 shows this process.

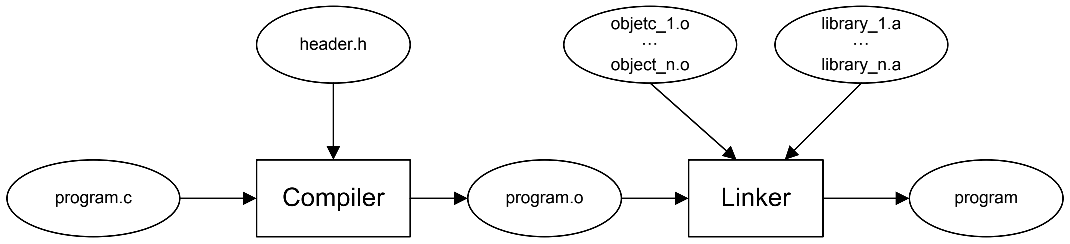


Figura 1.2: Program development cycle

An example is shown next. A function `circleArea` is created. This function returns the area inside a circle given its radius. This function is provided for being reused by other programs, so it is defined in a separate file from the main program.

- ❏ Edit `1-1circle.c` with the following content:

```
#include "1-1circle.h"

double circleArea(double radius)
{
  const double PI = 3.1415;
  return PI * radius * radius;
}
```

- ❏ Edit the header file `1-1circle.h`, where the prototype of `circleArea` is defined.

```
// Use always this statement in header files
#pragma once

// Function prototype
double circleArea(double radius);
```

  The statement `#pragma once` tells the compiler to add the header file to the compilation only once. It must be used always to avoid compile-time errors due to identifier duplication when referring the header file from several source files.

- ❏ Edit `1-1program.c` with the main program, which uses the `circleArea` function:

```
#include <stdio.h>

// Include the header file
#include "1-1circle.h"

int main()
{
  double radius = 3.0;
  double area = circleArea(radius);
  printf("Circle area (radius=%f) is %f\n", radius, area);
  return 0;
}
```

- ❏ Compile both files to get the object file:

```
$> gcc -c 1-1circle.c 1-1program.c
```

- ❏ Link the program telling gcc the object file where the code of `circleArea` is stored.

```
$> gcc 1-1circle.o 1-1program.o -o 1-1program
```

- ❏ Run the program.

The standard defining the C programming language also defines a function library which all compilers must include, called the C standard library. This library includes the code of the functions defined in the C standard.

A function included in the C standard library is `printf`. You can read the manual entry for this function and identify how it is called and what header file it needs.

❑ Open the manual entry of `printf`.

```
$> man 3 printf
```

❑ What header file is required to properly compile a program calling this function?[1]

The linker links all the programs with the C standard library by default. Thus, there is no need to specify any object file when calling the linker.

However, there are functions defined in other libraries. For example, the `sqrt` function, which computes the square root of a number.

❑ Read the manual entry of `sqrt`.

```
$> man 3 sqrt
```

❑ What header file is required?[2]

❑ Is it necessary to activate a flag while calling the linker?[3]

❑ Edit `1-1root.c`.

```c
#include <stdio.h>
// Add the required #include directive

int main()
{
  double num = 9.0;
  double root = sqrt(num);
  printf("The square root of %f is %f\n", num, root);
  return 0;
}
```

❑ Compile the program.

❑ Link the program. An error will occur since the `sqrt` function is not defined. The header file has being included, but it only contains the function prototype, not its code.

```
$> gcc 1-1root.o -o 1-1root
1-1root.o: In function ‘main’:
1-1root.c:(.text+0x2d): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

❑ Link the program with the linking option retrieved from the `sqrt` manual entry.

❑ Run the program.

It is of vital importance differentiating compiling errors from linking errors.

3.3. Automating the generation of the executable file

UNIX-like operating systems usually include a tool called `make` which allows automating the generation of executable files. To do so, a file usually called `Makefile` is used. This file contains a set of rules indicating how the executable file must be generated taking into account what files must be regenerated when others have been modified. When a file named A must be regenerated as a file named B is changed, it is said that A depends on B.

This tool provides a great advantage: it automatically checks whether the dependent files have been modified and only generates files that depend on these modified files, saving valuable time in the executable file building process.

The rules in a `Makefile` file associate a target with the required prerequisites, or dependencies, and the commands used to build this target. These rules are as following:

```
target : prerequisites
<Tab> commands
```

Note that a tab character must be written before the command that builds the target.

Next, an example using a `Makefile` file for building the `1-1program` file used above is shown.

```
# Rule for generating the main program
1-1program : 1-1program.o 1-1circle.o
  gcc 1-1program.o 1-1circle.o -o 1-1program

# Compile source files
1-1program.o : 1-1program.c 1-1circle.h
  gcc -c 1-1program.c

1-1circle.o : 1-1circle.c 1-1circle.h
  gcc -c 1-1circle.c

# Clean object files
clean :
  rm -f 1-1program.o 1-1circle.o
```

❏ Edit the `Makefile` file with the above content in the same folder as the files `1-1program.c`, `1-1circle.c` and `1-1circle.h`.

❏ Build the executable file by invoking the `make` command. You can specify the name of the executable file to be generated. If you do not specify any argument, the first rule of the `Makefile` file is executed; in this case this rule is the one that generates the executable file.

```
$> make
gcc -c 1-1program.c
gcc -c 1-1circle.c
gcc 1-1program.o 1-1circle.o -o 1-1program
```

- ❑ Try to generate the executable file again by using the `Makefile` file. The `make` command will check that the dependencies of the `1-1program` target have not changed; thus, it will detect that the executable `file` is up to date and does not require regeneration.

```
$> make
make: ‘1-1program’ is up to date.
```

- ❑ Delete the temporary files created while building the executable file by means of the `clean` rule of the `Makefile` file.

```
$> make clean
rm -f 1-1program.o 1-1circle.o
```

You can use this `Makefile` file as a template for the rest of the programs you will write in future lab sessions. You must be careful while editing the names of the files in the prerequisite sections of the rules.

3.4. Screen output

The `printf` function can be used to send a set of characters to the screen. This function is declared in `stdio.h`. The next sentence prints a character string on the screen:

```
printf("This string is shown on the screen");
```

This function can take several parameters. The first one is a format control string, which can contain:

- Text, which will be displayed on the screen as it is.
- Scape sequences, which start with the character '\' and are followed by another character. '`\n`' and '`\t`' represent a line break and a tabulation, respectively.
- Conversion specifiers, which are used to properly show the content of variables. They start with the character '`%`':

  `%c`: character (`char`).

  `%d`: signed decimal integer (`int` or `short`).

  `%f`: real (`float` or `double`).

  `%s`: character string (`char*`).

  `%x`: signed hexadecimal.

  `%p`: pointer.

The rest of the parameters of the function are the variables to be printed, which appear in the same order of the conversion specifiers.

❏ Edit `1-1output.c` with the following content:

```
#include <stdio.h>

int main()
{
  int i = 3;
  float f = 2.4;
  char * s = "Content";

  // Print the content of i

  // Print the content of f

  // Print the content of the string


  return 0;
}
```

❏ Complete the program so that it prints the following on the screen:

```
i content: 3
f content: 2.400000
s content: Content
```

❏ Compile and link the program using a `Makefile`. If you have any question regarding this step, ask your teacher for help.

❏ Run the program.

```
$> ./1-1output
```

3.5. Pointers

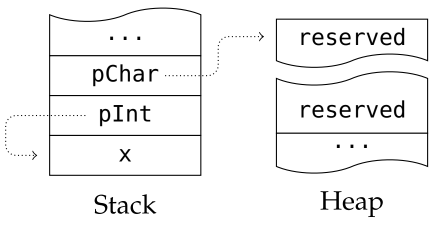
Pointers and their arithmetic are the most complex part of the C programming language, but also are the basic part for system programming since they allow low level memory management. A pointer is a variable that stores a location in memory, that is, a memory address. Thus, a pointer refers to a piece of data in memory. To define a pointer, the type of the memory that the pointer references followed by an asterisk must be used. Next some pointer definitions are shown:

```
char* pChar;  // Refers to a piece of data in memory 1-byte wide
int* pInt;    // Refers to an int in memory (usually 4-bytes wide)
double* pDouble1, * pDouble2;  // Refer to float numbers with double
                               // precision (8-bytes wide)
```

When defining a pointer as above, it cannot be immediately used since it points to an unknown memory area. It is necessary to initialize the pointer with a valid memory address to read from or write to it by means of the pointer. This can be done in two different ways: 1) making the pointer referring to the memory address of a program variable (by using the '&' operator to get the memory address of the variable), or 2) making the pointer referring to an area in memory dynamically allocated, commonly in the *heap*.

```
int main()
{
  int* pInt;
  int x;
  char* pChar;

  x = 3;
  pInt = &x;
  pChar = (char*)malloc(20 * sizeof(char));
  free(pChar);
  return 0;
}
```



The above example shows how memory is dynamically allocated in the heap by means of the `malloc` function. The `sizeof` operator returns the size (in bytes) of a data type. Furthermore, a casting is required when assigning the address memory to the pointer since the type of the value returned by the function is `void*`, which means "pointer to an unspecified type".

Unlike other languages that incorporate a garbage collector, such as Java, programming in C requires the program to free the memory allocated explicitly in the heap. The `free` function is used to do this.

The dereference or indirection operator '*' is used to operate with pointers. When this operator precedes the name of a pointer type variable, the content of the memory referenced by the pointer is accessed. The next statement added after the above code just before `return` will output the sequence `3 3` on the screen.

```
printf("%d %d", x, *pInt);
```

A very common use of pointers is passing parameters or arguments to a function. In C parameters are passed by value[6]. That means that if a variable is used as parameter, its value will be copied to a new variable inside of the function. If this value is changed inside of the function, it will not affect the value of the variable outside the function.

If a function needs to change the value of a variable outside of the function itself, it has to receive a pointer to the variable. The function now can change the memory pointed by the pointer (not the pointer itself) using the dereference operator.

The following snippet shows an example of passing a variable to a function:

```
#include <stdio.h>

void f(double d)
{
  d = 4;
}

int main(int argc, char* argv[])
{
```

[6]Note that in C++ functions can receive references to variables (for instance, the type of one parameter can be `int&`, that is, a reference to an `int`). However, references are not available in C.

```
  double d;
  d = 3;

  f(d);
  printf("%f\n", d);
  return 0;
}
```

❏ Edit the file `1-1pointer1.c` with the above code and check the value printed on the screen.

❏ Now, copy the above file into a new one: `1-1pointer2.c`. Then edit this new file so that, instead of receiving a `double`, the function receives a pointer to a `double`. Furthermore, you must also modify the assignment in order to modify the content of the memory location referenced by the pointer (use the indirection operator). Check that the program prints the value `4` in the screen. If not, ask your teacher.

3.6. Data structures

Structures arrange labeled objects, possibly of different types, into a single object. These labeled objects of an structure are stored consecutively in memory. The following program shows how to define and use a data structure called `Person`, which includes a string, an int and a double-precision float.

❏ Edit the file `1-1person.c` with the following content.

```
#include <stdio.h>

struct _Person
{
  char name[30];
  int  heightcm;
  double weightkg;
};
// This abbreviates the type name
typedef struct _Person Person;

int main(int argc, char* argv[])
{
  Person Peter;
  strcpy(Peter.name, "Peter");
  Peter.heightcm = 175;

  // Assign the weight

  // Show the information of the Peter data structure on the screen

  return 0;
}
```

❏ Insert the necessary statements to implement the actions described in the comments, so that when the program is executed it will show the following information on the screen:

```
Peter’s height: 175 cm; Peter’s weight: 78.7 kg
```

❏ Build the program and run it.

It is also possible to create pointers referencing data structures. In the above example, a new type of pointer to a `Person` could be defined.

```
// Type name abbreviations
typedef struct _Person Person;
typedef Person* PPerson;

int main(int argc, char* argv[])
{
  Person Javier;
  PPerson pJavier;

  // Memory location of Javier variable is assigned to the pointer
  pJavier = &Javier;
  Javier.heightcm = 180;
  Javier.weightkg = 84.0;
  pJavier->weightkg = 83.2;
  return 0;
}
```

As you can see, accessing the fields of the structure by means of the pointer requires using the ′`->`′ operator. For instance, the expression `pJavier->weightkg` is equivalent to `(*pJavier).weightkg`. That is, first, the content referenced by `pJavier` (which is a data structure of type `Person`) is accessed and, then, its `weightkg` field is accessed.

4. Files in your working folder

You must save all the files you worked on in this lab session in your memory stick.

5. Exercises

✏ Write a program named `1-1add.c` with a function named `add` that receives two arguments: an array of integers and its length. This function must return the addition of all the elements of the array. The `main` function of the program must be the written as follows:

```
#include <stdio.h>

#define NUM_ELEMENTS 7
```

```
int main()
{
  int vector[NUM_ELEMENTS] = { 2, 5, -2, 9, 12, -4, 3 };
  int result;

  result = add(vector, NUM_ELEMENTS);
  printf("The addition is: %d\n", result);
  return 0;
}
```

✏ Write a program named `1-1copy.c` to copy character strings by means of the function named `copy`. This function must have the following prototype:

```
int copy(char * source, char * destination, unsigned int lengthDestination);
```

The function receives the string to copy in the first argument. Character strings in C end with the character `0`. The second parameter is a pointer to the memory area where the string will be copied. The third parameter specifies the size (in bytes) of this area. This third parameter should be used by the function for not writing out of the memory reserved for the destination string.

SESSION 2

Analysis of computer performance using benchmarks

Objectives

In this lab session the concepts of measuring the execution time of programs running in GNU/Linux is described, with the main goal of analyzing the performance of the computer. To do so, the features provided by the operating system to access high performance counters of the computer are used. In addition to measuring the execution time of several programs used as synthetic benchmarks, the performance of several computers will be compared based on an application benchmark (real workload). Finally, the results obtained will be compared with the results provided by benchmark suites of organizations like SPEC (*Standard Performance Evaluation Corporation*).

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.
- Attend the lab session with a USB memory stick to store the files written while working on it.
- Be able of building and running programs written in C by using a terminal (command line interface) of the Linux operating system.

Session development

1. Source code instrumentation in GNU/Linux

Sometimes, measuring the execution time of a program, or a program fragment, is required. The measurements obtained can be used for multiple purposes such as comparing the performance of several systems or determining which is the portion of the program that consumes more time in order to optimize it.

Usually, the operating systems provide some high-level functions that allow accessing the time stamp counters (TSC) of the processor, also called performance counters. These counters count the number of ticks of the clock signal of the processor. Using the number of ticks between two events in the system, and taking the clock rate into account, a high-precision value for the elapsed time between the two events can be determined. Different operating system provides different methods to access these counters in the processor.

The POSIX API which Linux implements provides the following function (for C and C++) to obtain a time stamp from any clock of the system, such as the real-time clock (`CLOCK_REALTIME`).

```
int clock_gettime(clockid_t clk_id, struct timespec * tp);
```

Provides the time of the clock specified in `clk_id` and stores it in the variable referenced by `tp`. The return value is 0 in case of proper execution and -1 if an error occurs (in this case the `errno` variable is updated with the appropriate value identifying the error).

The `timespec` type variable is a data structure containing the following information, where `time_t` is an integer that depends on the system configuration:

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

This function can be used to precisely measure the elapsed time between two events in the system, as shown in the following snippet.

```
#include <time.h>

int main()
{
  struct timespec tStart, tEnd;
  double dElapsedTimeS;

  // Initialize the measuring task
  clock_gettime(CLOCK_REALTIME, &tStart);

  // Code to be measured
  ......
  ......
  // Finish the measuring task
  clock_gettime(CLOCK_REALTIME, &tEnd);

  // Compute the elapsed time, in seconds
  dElapsedTimeS = (tEnd.tv_sec - tStart.tv_sec);
  dElapsedTimeS += (tEnd.tv_nsec - tStart.tv_nsec) / 1e+9;

  return 0;
}
```

Notice that when computing the elapsed time in seconds, an integer-to-double casting is used, possibly generating rounding error. In this lab session this is not important (taking into account the small times measured) but it could be critical in some other context.

2. Features of the system under test

Before comparing the performance among several computers, the main features of each computer should be identified. This information can be obtained by means of the operating system or other programs. The hardware features of the computer will be listed using the `CPU-Z` program[1]. There is a copy of this program in the course DVD.

The information gathered by this program must be written in the spreadsheet of unit 1 results. This spreadsheet already contains the information of the reference computer, S_{ref}, or reference system, which will be used to compare the performance of another computer, SUT, or system under test. The SUT will be the computer of the lab.

- ❏ Run the `CPU-Z` program[2] and fill in the information regarding the features of the SUT in the spreadsheet.
- ❏ Try to obtain the name of the hard disk of the computer where the operating system is installed and write it in the spreadsheet.
- ❏ Complete the information in the spreadsheet with the version of the operating system which is running in the SUT. You can obtain this information using the system monitor: `Sistema`→`Administración`→`Monitor del sistema`→`Sistema`.

[1] http://www.cpuid.com

[2] Running this program requires administrator privileges in the operating system. If you have not this privilege on Windows, you can run this program using the `Wine` emulator in GNU/Linux (you must configure the emulator in Windows 98 emulation mode).

3. Performance analysis based on monolithic, synthetic workload

Synthetic benchmarks, or programs used to measure the performance of a computer based on synthetic workload, are programs that try to reproduce the more common operations performed by real applications. These programs may execute compilers or programming language translators, complex mathematical computations, or graphics processing amongst others, depending on the workload to be reproduced.

In this section of the lab session, synthetic benchmarks working with real numbers and data structures will be used. Thus, apart from the CPU of the system, the memory and the input/output system will be used. These benchmarks may be useful, for example, for comparing the performance of computers focused on scientific computations.

The `bench_fp.c` file contains the incomplete code of a program that accesses the high performance counter of the system in order to measure the response time of the `Task` function. This function is in charge of invoking the `DoFloatingPoint` function with different arguments. The `DoFloatingPoint` function allocates memory for three arrays of real numbers (using the `malloc` function), initializes two of them with random values (using the `rand` function), and performs several arithmetic operations over real numbers to initialize the elements of the third array.

Next, a brief description for each function that appears in this program is shown.

```
void * malloc(size_t size);
```

Allocates `size` bytes in memory and returns a pointer to the allocated memory. The allocated area in memory is not initialized. In case of an error the returned value is `NULL`.

```
int rand(void);
```

Returns a pseudo-random integer value in the range [`0`, `RAND_MAX`].

```
void srand(unsigned int seed);
```

Set `seed` as the seed for a new sequence of pseudo-random integers to be returned by `rand`. These sequences are repeatable by calling `srand` with the same seed value.

❑ In this snippet, how many times is the instruction `pdDest[i] = sqrt(pdDest[i])` executed in the whole program?[1]

1

❑ Fill the «Complete» sections of the program according to the scheme shown in the first section of this lab session.

- ❏ Using the command-line interface, go to the `bench_fp` directory. Compile and link the program with the `Makefile` provided.

  Run the `bench_fp` program with the following command:

  ```
  $> ./bench_fp
  ```

  If you have completed properly the missing statements in the source file, the elapsed time of the `Task` function is shown in the command-line interface.

- ❏ Since only one measurement may contain errors difficult to estimate, you must run the program four more times. Write down the elapsed times in the spreadsheet provided for this lab session (sheet `1-2`, columns `t_1` to `t_5`, row `bench_fp`). Compute the average elapsed time, the standard deviation and the confidence interval of the mean by programming the required expressions in the column `t_1-to-t_5 Average`, `t_1 a t_5 Std. Dev.` and `Confidence Interval of the Mean (95%)`, respectively.
- ❏ If the `Task` function is considered as the «task» to be executed by the system, which is the average throughput of the system expressed in tasks per minute? Write down the result in the spreadsheet in the column `Throughput (tasks/min)`.
- ❏ Compute the speedup of the SUT in relation to the reference system using `bench_fp` as the workload. Write down the result in the spreadsheet.
- ❏ Taking this program into account, together with the hardware features of the processor, do you think that this benchmark is really useful to measure the ability of the system for scientific computations? To check your answer show the `Resource view` (`Visor de recursos`) of the `System monitor` (`Monitor del sistema`) as follows and run again `bench_fp`.
  `Sistema`→`Administrador`→`Monitor del sistema`→`Recursos`
  Which is the percentage of CPU usage?[2]

2

4. Performance analysis based on multi-threaded, synthetic workload

The performance of the system executing the `bench_fp` program is limited by the performance of the system when it executes only one thread, regardless other factors such as the memory system that may also affect its final performance. In modern computers, processors consist of several cores (each processor die contains several processors assembled together). Furthermore, each core may provide several execution paths (such as the HyperThreading technology from Intel). Therefore, to get the maximum performance of a system, the paradigm of parallel programming must be used, or at least several instances of the same program must be executed.

Several basic concepts regarding programming with threads will be reviewed in order to improve the benchmark. A thread is the minimum unit that can be scheduled by an operating system. By default, a program, properly speaking, a process, consists of one execution thread. Thus, creating and managing multiple threads in a program relies on the programmer, or in the thread library used. The POSIX standard defines the API for creating and managing threads in an application.

Creating a thread requires calling the following function, which is defined in the header file `pthread.h`.

```
int pthread_create(
  pthread_t * thread, const pthread_attr_t * attr,
  void * (*start_routine) (void *), void * arg);
```

Creates in the calling process a new thread ready to be executed. The `thread` argument will contain the identifier of the new thread, whereas the `attr` argument is used to determine attributes for the new thread (usually this argument will be `NULL`). The `start_routine` argument is a pointer to the function to be executed by the new thread; `arg` is the argument for this function. If the function succeeds, the returned value is 0; otherwise, an error number is returned.

The main thread of the process, or any other thread, can wait until the end of another thread before continuing its execution. To do so, the following function may be used.

```
int pthread_join(pthread_t thread, void ** retval);
```

Waits for the thread specified in `thread` to terminate. The pointer returned by the terminated thread can be retrieved using `retval`; if the programmer is not interested in this value, he can specify `NULL` in this parameter. In any case, it must be taken into account that the returned pointer must reference memory locations valid after the thread termination; thus, usually these locations are in the heap (memory allocated by calling `malloc`). If the function succeeds, the returned value is 0; otherwise, an error number is returned.

Next, a simple example of a multi-thread program is shown.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void* ThreadProc(void* arg)
{
  const int TIMES = 6;
  int i;

  // Cast
  int n = *((int*)arg);
  for (i = 0; i < TIMES; i++)
  {
    printf("Thread %d, message %d\n", n, i);
    sleep(1);  // Sleep 1 second
  }
  printf("Thread %d finished.\n", n);
  return NULL;
}

int main(int argc, char* argv[])
{
  const int N = 5;
  const int TIMES = 3;
  pthread_t thread[N];
  int i;

  // Thread creation
  for (i = 0; i < N; i++)
  {
    if (pthread_create(&thread[i], NULL, ThreadProc, &i) != 0)
    {
      fprintf(stderr, "ERROR: Creating thread %d\n", i);
      return EXIT_FAILURE;
    }
  }

  for (i = 0; i < TIMES; i++)
  {
    printf("Main thread, message %d\n", i);
    sleep(1);  // Sleep 1 second
  }

  // Wait till the completion of all threads
  printf("Main thread waiting...\n");
  for (i = 0; i < N; i++)
    pthread_join(thread[i], NULL);
  printf("Main thread finished.\n");
  return EXIT_SUCCESS;
}
```

❏ Write a `Makefile` to compile and link `1-2thread.c` with the following command:

```
$> gcc 1-2thread.c -o 1-2thread -lpthread
```

❏ Run the program several times and try to understand what is happening in the screen. Can you explain it?

The folder `bench_fp_mt` contains a multi-thread version of the `bench_fp` benchmark. In the source file, the `NUM_THREADS` constant defines the number of threads to be executed in the program in addition to the thread that executes the `main` function. Each of these threads is created by calling `pthread_create`, which receives the memory address of the function to be executed, among other arguments. In this case, all the threads execute the same function: `Task`. Then, the main thread waits for the termination of the `NUM_THREADS` threads. This synchronization is required because, in C or C++, the whole program terminates when the `main` function is finished. Therefore, without this synchronization the program would terminate before the threads created to execute the `Task` function had finished their execution. This synchronization is done by means of the `pthread_join` function.

- ❑ How many times is the instruction `pdDest[i] = sqrt(pdDest[i])` executed in the whole program?[3] 3
- ❑ Complete the source file in the sections labeled as «Complete» according to the scheme shown in the first section of this lab session.
- ❑ Compile, link, and run the program (you can used the provided `Makefile` file if you want).
- ❑ Run the program four more times and write down the five measurements in the spreadsheet. Furthermore, compute the statistics for the response time of the program in the row `bench_fp_mt`.
- ❑ If the `Task` function is considered as the «task» to be executed by the system, which is the average throughput of the system expressed in tasks per minute? Write down the result in the spreadsheet in the column `Throughput (tasks/min)`. Is the increase in the throughput lower, higher or equal to the increase in the number of cores of the processor?[4] 4
- ❑ Compute the speedup of the SUT in relation to the reference system using the `bench_fp_mt` program as the workload. Write down the result in the spreadsheet.
- ❑ Run the program once more while observing the performance view provided by the `System monitor` of the operating system. Which is the percentage of CPU usage in this case?[5] 5

5. Performance analysis based on real workloads

Benchmarks based on real workloads, also called application benchmarks, execute real applications for comparing the performance among several systems. These applications should represent the final workload of the system for the comparisons to be useful.

In this section, a real application will be used for comparing the performance between two systems. This application is Mencoder[3] and it is used for encoding and decoding video files.

[3] http://www.mplayerhq.hu

- ❏ Open the system monitor by means of:
  `Sistema`→`Administración`→`Monitor del sistema`.
- ❏ Open a command-line interface and access the `mencoder` directory. Run the script `x264` asking the operating system to show its elapsed time.

  ```
  $> time ./x264
  ```

  You must bear in mind that you need to provide the required permissions to execute this script. As studied in the previous lab session, this is done as follows:

  ```
  $> chmod +x x264
  ```

  This script invokes Mencoder for rescaling a video sequence using the x264 codec. The script asks `mencoder` to use only one thread, so the system monitor should report that only one CPU is working 100 %. You can see the content of the script typing the following command:

  ```
  $> more x264
  ```

  Run the script five times and compute the statistics for the response time of the application. Write down the result in the spreadsheet. In addition, compute the throughput, that is, how many video files per minute can the system encode, as well as the speedup of the SUT in relation to the reference system

The implementation of the x264 codec is multi-threaded. Thus, the same rescaling task will be now done using as many threads as the processors of the system, so the task is divided into several threads.

- ❏ Run, from the command-line interface in the `mencoder` directory, the following command:

  ```
  $> time ./x264_mt
  ```

  You may notice that all the processors of the system are being used. Run the script five times and compute the statistics for the response time of the application. Write down this result in the spreadsheet. In addition, compute the throughput of the SUT and its speedup in relation to the reference system
- ❏ In the Mencoder web page, it is stated that the multi-threaded implementation «[...] *increase[s] encoding speed linearly with the number of CPU cores (about 94% per CPU core), with very little quality reduction* [...]»[4]. Does this statement match the measurements you have taken? $\boxed{6}$

6

Once you have the speedup values for the SUT in relation to the reference system in the spreadsheet for `bench_fp`, `bench_fp_mt` and `mencoder`, a speedup

[4] http://www.mplayerhq.hu/DOCS/HTML/en/menc-feat-x264.html

ratio of the SUT in relation to the reference system can be computed taking all these results into account. Although we could think of the arithmetic mean of all the results as the speedup for all the applications, the geometric mean results more appropriate when comparing normalized numbers, such as performance ratios. Given n values $x_1, x_2, \ldots, x_n$, the geometric mean is computed as follows:

$$G = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n} = \left(\prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

❏ Compute, by using the geometric mean, the speedup ratio of the SUT in relation to the reference system and write down the result in the `C17` cell of the spreadsheet.

6. Performance analysis based on benchmark suites

Standardized collection of programs have been designed for comparing the performance among systems in very different environments, such as scientific, industrial, academic, etc. These collections are called benchmark suites. Its main goal is to be representative of the final workload of the system. The best known benchmark suites are defined and maintained by the SPEC (*System Performance Evaluation Corporation*)[5].

Among all the benchmark suites developed by SPEC, the best known and the most commonly used is SPEC CPU (its current version was released in 2006: SPEC CPU2006[6]). It is used to measure the performance of CPU-intensive and I/O-non-intensive applications. This suite includes real applications that have been migrated to most of the modern common platforms. It consists of 12 programs which operate over integer numbers (SPECint2006 or CINT2006) written in C and in C++, and 17 programs which perform operations over floating point numbers (SPECfp2006 or CFP2006), written in C, C++, and Fortran.

Both CINT2006 and CFP2006 provide two types of metrics: SPECspeed and SPECrate. The former measures response time whereas the latter measures throughput. Furthermore, each of these two metrics provides two values: base and peak. The first one is computed by means of the geometric mean of the provided results (either 12 integer or 17 floating-point tests) when the programs are compiled with normal optimizations (you will review compile optimizations in the next lab session). The second value is computed as the geometric mean of the results provided by the tests when these tests are compiled with optimizations for maximizing performance. Therefore, the results provided by SPEC CPU2006 for each system are the following ones:

- SPECint2006:

[5] http://www.spec.org

[6] http://www.spec.org/cpu2006/Docs/readme1st.html

    - SPECmetrics:
        - SPECint2006 (*peak*)
        - SPECint_base2006
    - SPECspeed:
        - SPECint_rate2006 (*peak*)
        - SPECint_rate_base2006

- SPECfp2006:
    - SPECmetrics:
        - SPECfp2006 (*peak*)
        - SPECfp_base2006
    - SPECspeed:
        - SPECfp_rate2006 (*peak*)
        - SPECfp_rate_base2006

The SPEC CPU2006 suite uses as reference system a Sun Ultra Enterprise 2 (UltraSPARC II processor @ 296 MHz), in which both SPECint2006 and SPECfp2006 ratios are 1.0. One system with a SPECint2006 ratio of 2.0 is twice as fast as an Sun Ultra Enterprise 2 when performing operations over integers.

- ❏ Go to the SPEC web page and look for the results published for the SPEC CPU2006 suite.
- ❏ Look for the results of a system using a processor similar to the one used in the PC of the laboratory (the web provides a search engine to filter the results by hardware components, such as the processor, the motherboards, etc.).
- ❏ Once a similar system is found, you may see the results of each test (CINT2006, CFP2006, CINT2006 rates and CFP2006 rates) in several formats: HTML, CSV, PDF, etc. Choose CINT2006 in PDF, for example. Notice how the results are presented. For each test, either integer or floating-point numbers, the speedup of the SUT in relation to the reference system is shown. You can check how the final ratio, for instance SPECint_base2006, is the geometric mean of the speedup obtained by each individual test.
- ❏ Write down the ratios for the SUT asked in the spreadsheet. Are the speedup ratios of the SUT you computed previously in this lab session comparable with the ratios provided by SPEC?

7. Files in your working folder

You must save all the files you worked on in this lab session in your memory stick: `bench_fp.c` and `bench_fp_mt.c` completed, as well as the spreadsheet you must deliver `unit1.xls`.

8. Exercises

✏ It is highly recommended for you to read the description of the SPEC CPU2006 benchmark suite, since it is one of the most commonly used benchmarks these days. You can access this description by following this link:
`http://www.spec.org/cpu2006/Docs/readme1st.html`

SESSION 3

Analysis of program performance

Objectives

In this lab session the student is introduced to analysis of program performance by using profilers, which are tools that analyze program performance. Amdahl's law will be applied for guiding the optimization process and for determining what is the maximum improvement that can be achieved with this process. In addition, the student will see how architecture features impact program performance.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.
- Review Amdahl's law.
- Attend the lab session with the spreadsheet that was begun in the previous lab session.

Session development

1. Introduction to analysis of program performance

❏ Boot the computer with the course's Linux distribution and keep on reading while the computer boots.

The performance of a program is one of its fundamental characteristics. The optimization process tries to improve this performance, but should be conducted in a rational manner. However, «optimizations» (changes in the code that the programmer believes will make the program more efficient) are many times carried out intuitively and, in fact, they do not increase the performance while, at the same time, go against other important code features, such as clarity, readability and maintainability. A famous quote by Donald Knuth[1] emphasizes this problem: «premature optimization is the root of all evil».

Premature optimization is that one which tries to optimize before knowing which parts of the code have real importance in its execution time. The analysis of algorithms is a very useful tool for comparing the theoretical performance of different algorithms. However, real programs have a large source code, are executed on very complex computer architectures and go through a compilation phase that can lead to optimizations which are not obvious in the high-level language; all this means that the analysis of algorithms is not enough. The programmer should also measure the program execution to really know what needs to be optimized.

Program execution measurement can be carried out in several ways. As you have already done in the previous lab session, the code can be manually instrumented, that is, probes can be inserted in the code for taking the time at that point. The execution time of a program fragment can be obtained using the difference between the time at the beginning and at the end of the fragment. If the time stamp taken by these probes is based on the system time, we obtain what is called *wall-clock time*. In a multi-tasking system, this time will not depend only in the execution of the code by itself: it can also be affected by the execution of other programs. Logically, for optimization, the time really consumed by the program is the only time that counts. Therefore, the measurement should be carried out when less elements can interfere with it. It also should be repeated several times to reduce the significance of random errors.

This measurement technique is very useful. A good practice that many programs follow, especially in server environments, is including probes in the program to log events during execution. This is very helpful for debugging and also for knowing which parts of the program are taking most of the time. This log is generally enabled "in production", i.e., when the program is normally used. This provides very valuable data about program execution in real-life conditions. However, these measurements are usually of large chunks of code: if many probes were used, the measurement code would slow the application.

In order to know precisely how long each code fragment takes, some specific tools have been developed to be used in the final phase of development and not in production. The most important tools are *profilers*. They obtain a profile of the program execution. This profile is made up of several data about the program as, for example, how many times a code fragment (usually a function, although there

[1]Donald Knuth is one of the most recognized experts in computer science. His book series *The Art of Computer Programming* are fundamental books. He is also known as the father of the analysis of algorithms and he is the developer of TEX, a computer typesetting system used in many books, magazines and these lab handouts.

are profilers that go to the line level) is executed, how long it takes on average to execute it, etc. After obtaining this data, appropriate optimization decisions can then be taken.

Profilers usually can generate two kind of profiles:

- Flat profile: shows how long each function takes on average and how many times each function is called.
- Call graph: shows how many times a function was called from another function, which helps in identifying relationships between functions.

The profilers' goal is pointing to hotspots, that is, zones in the program that take a lot of time. Following Amdahl's law, these spots are the most interesting for optimizing because they will be the ones that yield a higher speedup.

This lab will demonstrate the process of performance analysis of programs with small programs. Logically, the more complex a program is, the more difficult this process is and the more useful optimizations are.

2. Performance analysis of C programs

2.1. Study of the program to analyze

Before using a profiler, we are going to review the code that we will analyze.

- ❑ Download from the Virtual Campus the files corresponding to this lab session.
- ❑ Open `prog1.c` with an editor.
- ❑ Find the `main` function at the end of the code. As you can see, this function declares two arrays and then calls another function to fill in the arrays with random numbers. Afterwards, it calls the `countSharedPrimes` function, which counts the primes that appear in both arrays. Finally, `main` calls a function that sorts the array.
- ❑ Study the code of the `countSharedPrimes` function. As you can see, it loops over the first array and, for each of its elements, checks if the element is in the other array and is a prime number. The `isInArray` function just goes over an array until it finds the number received as input and, if it does not find it, returns `FALSE`. The function `isPrime` goes over all the numbers between two and the number to analyze, checking if there is any division where the reminder is zero. If this happens, the number will not be prime. If all the divisions have a remainder different from zero, the number will be prime.

❏ Study the code of the `arraySort` function. This function sorts the array following the selection sort method.

❏ Compile the program with this command:

```
$> gcc prog1.c -o prog1
```

This command uses GCC, the C compiler developed by GNU, to compile and link the file `prog1.c` in just one step. The parameter `-o prog1` indicates that the output file should be called `prog1`.

❏ Run the program measuring how long it takes to execute with the following command:

```
$> time ./prog1
```

❏ Repeat the measurement five times and write down the resulting values in the row `Prog1 without optimization` of sheet `1-3`. Calculate the average, the standard deviation and the confidence interval of the mean in the corresponding cells. Which is the average execution time in seconds?[1]

1

Next, to see how difficult it is to optimize without measurements, we will make some assumptions and some calculations and then we will check to what extent we succeeded.

Which do you think is the most time-consuming function?[2] Write down the value in cell `B14`. What percentage of the execution time do you think that can be attributed to that function?[3] Write down the value in cell `B15`.

2

3

Imagine that you can optimize the code in that function so that it takes half the time. What speedup would you get in that function?[4] Write the value in cell `B16` What will be the speedup for the whole program?[5] Write the value in cell `B17`. What would be the program execution time?[6] Write the value in cell `B18`.

4

5

6

To check what are the real values, we will use several *profilers*.

2.2. *Profiling* with GProf

First, we will show how profiling works using `gprof`, a free tool with GNU's General Public License (GPL). In summary, the process is as follows:

- Recompiling the program with `-pg`, an option which tells the compiler to include the code for the profiler.
- Running the program. The execution will generate a profile, which is a binary file called `gmon.out`. If this file already existed, it would be overwritten.
- Use the program `gprof` to show as text the information in `gmon.out`.

In the case of your program, you must do the following:

- Recompile the program adding the profiler with this command:

  ```
  $> gcc prog1.c -pg -o prog1
  ```

  Notice that `-pg` has been added.

- Run the program with the following command:

  ```
  $> time ./prog1
  ```

  You shouldn't notice a significant change in the program execution time with respect to previous executions. If this happened, it would mean that the instrumentation is changing significatively the program execution.

- List the files in the directory to check if the `gmon.out` file has been generated:

  ```
  $> ls
  ```

- Generate the profiling report with the following command:

  ```
  $> gprof ./prog1 > report.txt
  ```

  Notice that you do not need to tell `gprof` that you are using the `gmon.out` file but you need to tell it which executable file was used to generate the `gmon.out` file. The last part of the command, `> report.txt`, is used to redirect the output of `gprof` from the screen to the `report.txt` file.

- Edit the `report.txt` file.

`gprof` output is divided in two parts: the flat profile and the call graph. In the first one you can see a line for each function. In each line, the columns have this meaning:

- `% time`: percentage of the time that the program spends in that function.
- `cumulative seconds`: cumulative total number of seconds the computer spent executing this function, plus the time spent in all the functions above this one in this table.
- `self seconds`: number of seconds accounted for this function alone. The flat profile listing is sorted first by this number.
- `calls`: total number of times the function was called.
- `self ms/call`: average number of milliseconds spent in this function per call.
- `total ms/call`: average number of milliseconds spent in this function and its descendants per call.
- `name`: name of the function.

After the explanation for each column, you can find the call graph. This graph is divided in sections with dashed lines. In each section there is a line with a number in brackets to the left. The section gives information about the function in that line. The previous lines are the functions that call this function, while the following lines are the functions that this function calls. For instance:

```
index % time    self  children    called     name

[...]

                0.00    0.32       1/1          main [1]
[3]      2.1    0.00    0.32       1        countSharedPrimes[...]
                0.27    0.00  100000/100000      isInArray[...]
                0.05    0.00  100000/100000      isPrime(int) [5]
-----------------------------------------------
```

gives information about the `countSharedPrimes` function, which is called by `main` and calls `isInArray` and `isPrime`.

For the main function in the section, the columns have this meaning:

- `% time` shows the percentage of the time that the function and the function it calls take from all the time taken by the program.
- `self` shows the total time in seconds spent in the function, without taking into account the time spent in the functions that it calls.
- `children` shows the time spent by the functions that are called by this function.
- `called` shows the total number of times that this function has been called.

Now that you have measured the application with the profiler, you can check if your assumptions were right:

- ❏ Which is the function that uses more time?[7] Write the value in cell `C14`. 7
- ❏ What percentage of the time does it use?[8] Write the value in cell `C15`. 8
- ❏ If you decreased its execution time in half, what would be the program execution time?[9] 9

The profiler can show where a program spends time, but no how to solve it. You need to analyze why this happens: is that function using a very complex algorithm that could be substituted by a more efficient one? Is that function being called too many times and what needs to be changed is the algorithm in the function that makes those calls? Could the function call more efficient instructions? Could it use more efficient data structures?

In this simple case, knowing that selection sort is one of the worst sorting algorithms, the first solution could be change it for quicksort. This is done in `prog2.c`. Let us check that it works:

- ❏ Compile and measure five times the execution time of this new program version. Write these value in the row `Prog2` of the spreadsheet. What is the average?[10] Write this value in cell `G4`. What is the program speedup?[11] Write this value in cell `H4`.

10

11

A new performance analysis should be carried out now to speed up the program even more. Compile with `-pg` the program `prog2`, run it and get with `gprof` its profile. Which function uses now the biggest percentage of time?[12] What percentage of time does it use?[13]

12

13

2.3. *Profiling* with KCachegrind

GProf is a tool with a rudimentary interface. An alternative is using KCachegrind, a display tool that uses the data obtained by Callgrind, which is a tool that uses the Valgrind instrumentation system. Valgrind's main objective is analyzing program memory usage, but when it is combined with Callgrind, it becomes a profiler.

Next, we are going to analyze `prog2` with KCachegrind:

- ❏ Compile the program with this command:

  ```
  $> gcc prog2.c -o prog2
  ```

  Notice that there is no need to include a special compilation option for KCachegrind. In fact, if you include `-pg`, there are conflicts between the two tools and they can lead to execution errors.

- ❏ Run the program using Callgrind inside Valgrind with this order:

  ```
  $> valgrind --tool=callgrind ./prog2
  ```

  As you will see, running a program with Valgrind takes much longer than a normal execution. This is its great disadvantage compared to GProf. The reason is that GProf uses temporal sampling: every so often, it looks what function is running. Valgrind, instead, simulates certain aspects of the processor, as the cache memory, and instruments all function calls.

- ❏ Check that the `callgrind.out.pid` file has been generated in the working directory. Take into account that `pid` is a number that identifies the process executed and will change in each execution. Use the following command:

```
$> ls callgrind*
```

❏ View the profile with KCachegrind using this command:

```
$> kcachegrind callgrind.out.pid
```

The KCachegrind window has two main areas:

- Left: it shows a flat profile with the function list and information such as the percentage of time they used (self and including children) or the number of times each function was called. Clicking on the column headers you can sort the list by the corresponding criterion.
- Right: it can show different data about the function selected on the left panel. One of the most interesting views is *Callee Map*, which shows two views: in the top view, it depicts each function with a rectangle with area proportional to the time used; in the bottom view, it shows a call graph.

Let us use KCachegrind to analyze the program:

- Select the `main` function in the left part if it was not already selected.
- Selecting *Callee Map* in the top right zone, you can see a very large rectangle for the function that takes the biggest amount of time that, as GProf told us, is `isInArray`. Inside the rectangle you have the percentage of time used by the function. What is it?[14] Write it in cell `B23`. It will probably not be the same that GProf gave, but if both profilers are working correctly, they should be very similar. 14
- Select *Call Graph* in the bottom right to see the call graph. You can see there which functions are called by `main`. Note that there are some functions over it: they are part of the C runtime execution environment, which was included by the compiler. What percentage of time is used by `main`?[15] Write it in cell `B24`. 15
- Double click on the rectangle of function `countSharedPrimes`. As you can see, only the part of the call graph corresponding to this function is shown now and other functions (for instance, `sort`) have disappeared. In addition, the value shown for the `main` function has changed. What is this value now?[16] If you compare it with the previous answer, you can see that it is smaller. This is because the cost of `main` shown is only the cost corresponding to the part of `main` that is executing the function selected below, which is `countSharedPrimes` in this case. 16
- Next to the arrows that connect two nodes in the graph, you can see the number of calls that the node in the top makes to the node in the bottom. How many times does `main` call `countSharedPrimes`?[17] Write it in cell `B25`. 17

From this graph, we can infer that most of the program execution time is spent in `isInArray` directly, as it does not call any other function. For improving the performance we should try to improve the function code or we should try to call it fewer times. A possible solution could be sorting the array before calling `countSharedPrimes` so that later we could use the fact that the array is sorted and implement a binary search in `isInVector`. In this lab session, we are going to stop the analysis now, so you can close KCachegrind.

2.4. Influence of compiler optimization in performance

A high-level language instruction can be translated to machine code in many different ways, some with better performance than others. Modern compilers include an optimization phase that can be controlled with options. Enabling optimization has some disadvantages:

- The compilation process can take longer, as the compiler needs to do more work. In projects with a large amount of code, this can mean lost productivity for programmers if they have to wait a long time in each compilation.
- The compiler, just like human beings, can be wrong thinking that certain changes will make the code more efficient.
- Optimizations can make changes in the generated code so deep that it is difficult to connect machine code with source code. This hinders debugging the source code line by line and makes profilers' results less intelligible.

These reasons may advise against using optimization in some cases, but in general enabling them is better, at least the most basic ones to improve the performance of the final code.

From the point of view of the optimization process using profilers that we have seen, you should be aware that it is not worth optimizing by changing the source code when the compiler can obtain the same optimization transparently for the programmer for two reasons: 1) programmer time would be wasted in doing something that can be automatically done by compilers, and 2) as we have just mention, many times optimization makes the source code less intelligible. In any case, there are optimizations, especially at the highest level (for instance, changing the algorithm as we did in the previous example) that the compiler can not carry out. The best practice is measuring the code with compiler optimizations and only then, if we did not get the expected performance, proceed to apply source code optimization carried out by the programmer.

Let us see next the effect of compiler optimizations. In the examples that we have seen until now, no optimization was enabled. Let us repeat some experiments enabling optimizations to see the differences.

- ❏ To enable the most basic optimization level, you have to use the parameter `-O1` (the `O` comes from «Optimization»[2]). Compile the `prog1.c` program with this command:

  ```
  $> gcc -O1 prog1.c -o prog1
  ```

- ❏ Run the program five times taking times with this command:

  ```
  $> time ./prog1
  ```

- ❏ What is the average?[18] What speedup do you get comparing to the original execution?[19] Write these values in the row `Prog1 -O1` of the spreadsheet. Is the response time smaller than the response time of the version without optimizations?[20]

  18 19 20

- ❏ Compile now using the second optimization level with this command:

  ```
  $> gcc -O2 prog1.c -o prog1
  ```

- ❏ Run the program five times taking times. What is the average?[21] What speedup do you get comparing to the original execution?[22] Write these values in the row `Prog1 -O2` of the spreadsheet. Is the response time smaller than the response time of the version with `-O1`?[23]

  21 22 23

- ❏ Finally, repeat the process with the highest optimization level:

  ```
  $> gcc -O3 prog1.c -o prog1
  ```

- ❏ Run the program five times taking times. What is the average?[24] What speedup do you get comparing to the original execution?[25] Write these values in the row `Prog1 -O3` of the spreadsheet. Is the response time smaller than the response time of the version with `-O2`?[26]

  24 25 26

As you can see, the optimization process is not simple. So far we have tested the following options:

a) Compiling the original program with `-O1`.

b) Compiling the original program with `-O2`.

c) Compiling the original program with `-O3`.

d) Using `prog2`, the hand-optimized program.

Analyzing all these results, which is the fastest option?[27] However, we still have to see what happens if the compiler optimizations are used in `prog2`. Run the program with the three optimization levels. Which is faster?[28]

27 28

[2]You can get all the details about the optimization options that GCC offers with the command `man gcc`.

It is possible that `-O3` is slower than `-O1`. In order to understand this, the code generated by GCC should be carefully studied. This is beyond the scope of this lab. However, and just to see the complex ways in which architecture influences on performance, we will present a possible explanation: the level `-O3` applies optimizations that, in principle, represent an improvement in execution time but increase the size of the executable file. One consequence of this is that a piece of code (for instance, a loop) that previously could be held within the processor cache now does not fit and has to be fetched from a higher memory level. As we will discuss in the memory unit, this means a very high penalty for execution time. In additional exercises at the end of this lab this problem is studied further.

Note that the results obtained in this lab may be different from what you will see in actual programs that, in general, are much more complex. The most important lesson is that the optimization process is not easy because, due to the different abstraction layers between high level and the architecture that runs the programs, there are sometimes counterintuitive results; therefore, you should measure your code and use tools as profilers and compiler optimizations, but you should not blindly trust them.

2.5. Influence of compiler optimization in profilers

As noted above, compiler optimizations can affect the profiling process. Let us check it:

- ❏ Compile `prog1` with the maximum optimization level and the profiler code at the same time:

  ```
  $> gcc prog1.c -O3 -pg -o prog1
  ```

- ❏ Run the program to obtain `gmon.out`:

  ```
  $> ./prog1
  ```

- ❏ Generate the profiling report:

  ```
  $> gprof ./prog1 > report.txt
  ```

- ❏ Edit the file `report.txt`. How many functions can you see?[29] Write the value in cell `B30`. The reason for this is that one of the compiler optimizations is, instead of calling a function, copying all the code of the function where the call was. This avoids a `call`, which is usually an expensive operation. 29

- ❏ Repeat the same process using `-O2`. How many functions can you see?[30] Write the value in cell `B31`. Do you see in the profile all the important functions?[31] 30 31

3. Files in your working folder

At the end of the session, you must copy the file `unit1.xls` in your memory stick.

4. Exercises

- ✏ GCC's `-Os` optimization level gives more priority to optimized code that does not increment the object code size. To achieve this, GCC uses only the optimizations of level `-O2` that do not increment the object code size. Test `prog1` with this option. Do you get better performance than with `-O1`? And than with `-O2`? Compare the size of the executable files obtained with different optimization options. Which is the biggest? And the smallest?

- ✏ As previously noted, the optimization level, in addition to changing the execution time of the program, affects the compilation time, that is, the time needed by the compiler to do its task. To check it, measure the compilation time of `prog1` without optimizations and with the three optimization levels. What is the speedup of not using optimizations compared to each level?

- ✏ C's operator `&&` carries out a lazy evaluation, which means that it evaluates in a sequential order the sub-expressions that make up an expression and, when it founds one that is false, it stops evaluating the rest because the result will be false anyway. This is, in theory, an optimization. However, taking into account modern computer architectures that try to execute several instructions at the same time and suffer a penalization when they have to cancel their execution, it can result in lower performance.

  In program `prog2`, the execution time is dominated by the time spent in the function `countSharedPrimes`, which has a conditional statement that calls two functions: `isInArray` and `isPrime`. Using a profiler, determine if changing the order of the function calls inside the expression changes the number of calls to each function and if it changes the program performance.

- ✏ Sometimes, profilers give wrong information. Read the paper *Evaluating the accuracy of Java profilers*[3], where the authors explain how they used four different profilers on the same code and only two of them pointed to the same function as the one that consumed most of the time. This demonstrates that at least two profilers were giving wrong data (and it is possible that all four of them were wrong!).

[3] `http://sape.inf.usi.ch/publications/pldi10`