

1. INTRODUCTION

1.1. What does this program do?

This program periodically detects state modifications by calling command (executable file) named "detector" with its specified arguments.

1.2. About the files of the project

To make the program much better in understanding and to make the code itself less redundant, we decided to divide it into several modules. There are following files in our project:

run.c: this file contains the *run* function that executes provided command with its specified options provided as arguments to the function, and the function for printing these attributes just to verify whether they are working correctly or not.

run.h: header of run.c: Here you can also find small descriptions about the functions.

buffer.c: this file is just for controlling the buffers, i.e creating, comparing, copying and destructing buffers.

buffer.h: header of buffer.c: The special thing about this header is that there's a new structure *Buffer* which has the attributes *data* of type *char** and *size* of type *int* which holds the size of the *data*.

main.c: this is the main file, where we just define the general attributes: arguments with their specified options and commands. In this file one can find the parsing of options and its arguments with usage of *getopt* function.

Makefile: It enables quick compilation of the program via command "make". There are several targets in the Makefile:

- 1) all - will create executable file "detector".
- 2) clean - will clean all useless temporary or object files.
- 3) coverage - compile the sources in a special way with some additional arguments to measure the code coverage.
- 4) test - runs all the tests without valgrind.
- 5) test-sans-valgrind - the same as test.
- 6) test-avec-valgrind - runs the tests with valgrind.
- 7) gcov - generates the coverage reports. Be aware that before this the targets coverage and test should be performed.
- 8) couverture-et-tests - shortcut for coverage+test+gcov.

Test cases: There are 6 test cases for being sure that program is conform to specifications and don't produce any problems (regarding to memory, options, etc).

coverage.html: the html file which shows the result of coverage report. To get it, You can type in terminal

gcovr -r . --html -o name.html.

Be aware that gcovr should be installed in your machine.

link.txt: It contains the link for Overleaf documentation.

2. USER MANUAL AND PROGRAM DESCRIPTION

2.1. Manual

2.1.1. Compilation

For compiling the program, there are two ways: First way is to compile all 3 source files together in terminal using `gcc run.c buffer.c main.c -o detector`, after which it will create executable file `detector`.

Second way is to use the given Makefile. For this, we will just have to write **make** in terminal, after which it will itself create the executable file under the same name.

2.1.2. Usage

2.1.2.1) Syntax

To use this program you must write name of the program (i.e. `./detector`) and specified arguments in the terminal. We will refer to the given arguments using special functions, such as *getopt*, *execvp* and etc. The format of usage will be as follows:

`./detect -c -i interval -l limit -t format cmd args...`

2.1.2.2) Arguments

The argument *format* of option `-t` is the time with its specified format; The argument *limit* of option `-l` is the limit of launches; The argument *interval* of option `-i` is interval (in milliseconds) between two successfully executed commands; The option `-c` is for printing the exit code after each launch; And the last bunch of arguments *cmd args...* is the given command with its arguments to be launched.

2.2. How does the program work?

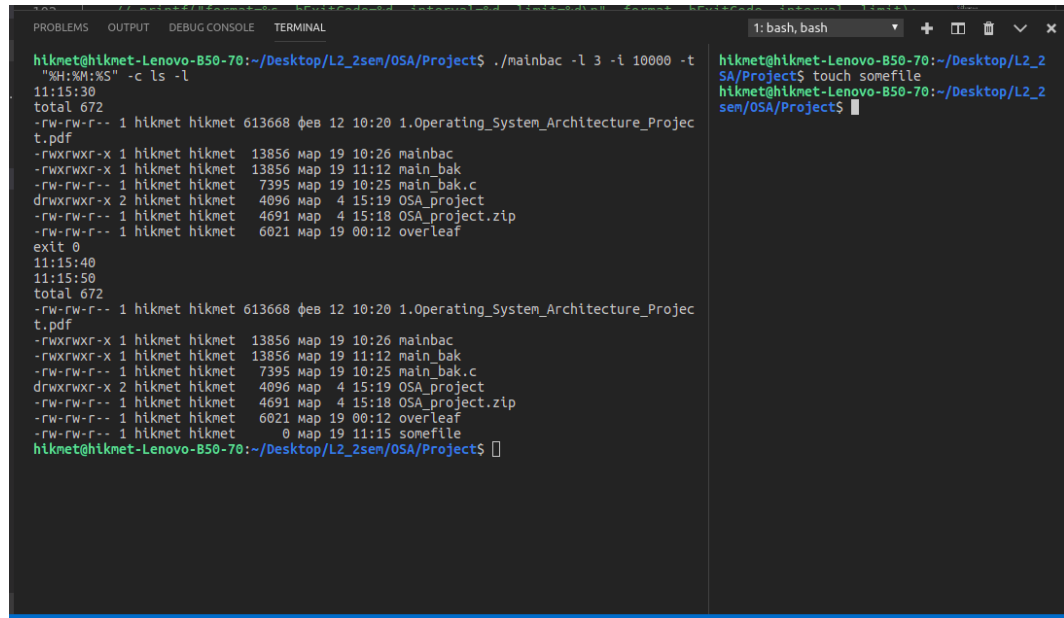
The program tries to run the provided command (possibly with some arguments) using the function *execvp*. The output of the first execution is saved in the variable *buf_old* of type *Buffer* that we have defined ourselves. Then starting from the further executions, the output is saved in the variable *buf_new* of the same type and is compared to *buf_old*'s data. If there's some difference, the *buf_old* will be changed and hold the contents of *buf_new*. We used pipe to be able to save and to write the output to buffers. Between two executions, there's some interval (by default or provided as argument) which is implemented using the function *usleep*.

Note: Since the code is well commented, You can find of some useful explanations in the files as well.

2.2.1. Expected output

After execution, on the screen each **interval** seconds **limit** times will appear formatted time, written command state, exit code (if they are provided by the user). But there is an important point: after each launch, it will show the next launch of command only if there will appear some changes during run-time. Otherwise it will just print each time the formatted time of every launch. The same thing for exit code.

2.2.2. Examples of successful result



```
hikmet@hikmet-Lenovo-B50-70:~/Desktop/L2_2sen/OSA/Project$ ./mainbac -l 3 -i 10000 -t
"%H:%M:%S" -c ls -l
11:15:30
total 672
-rw-rw-r-- 1 hikmet hikmet 613668 feb 12 10:20 1.Operating_System_Architecture_Projec
t.pdf
-rwxrwxr-x 1 hikmet hikmet 13856 map 19 10:26 mainbac
-rwxrwxr-x 1 hikmet hikmet 13856 map 19 11:12 main_bak
-rw-rw-r-- 1 hikmet hikmet 7395 map 19 10:25 main_bak.c
drwxrwxr-x 2 hikmet hikmet 4096 map 4 15:19 OSA_project
-rw-rw-r-- 1 hikmet hikmet 4691 map 4 15:18 OSA_project.zip
-rw-rw-r-- 1 hikmet hikmet 6021 map 19 00:12 overleaf
exit 0
11:15:40
11:15:50
total 672
-rw-rw-r-- 1 hikmet hikmet 613668 feb 12 10:20 1.Operating_System_Architecture_Projec
t.pdf
-rwxrwxr-x 1 hikmet hikmet 13856 map 19 10:26 mainbac
-rwxrwxr-x 1 hikmet hikmet 13856 map 19 11:12 main_bak
-rw-rw-r-- 1 hikmet hikmet 7395 map 19 10:25 main_bak.c
drwxrwxr-x 2 hikmet hikmet 4096 map 4 15:19 OSA_project
-rw-rw-r-- 1 hikmet hikmet 4691 map 4 15:18 OSA_project.zip
-rw-rw-r-- 1 hikmet hikmet 6021 map 19 00:12 overleaf
-rw-rw-r-- 1 hikmet hikmet 0 map 19 11:15 somefile
hikmet@hikmet-Lenovo-B50-70:~/Desktop/L2_2sen/OSA/Project$
```

Figure 1: program execution

Here you can see the execution and running process. As you see, on the first execution it prints content of entered command `ls -l`, then next 10 seconds it's just doing nothing, as there is no change. And then you can see in the right part of the terminal, that we create some file, so there appears the changes in command execution and prints the modified output in the third final execution.

2.3. Possible errors

There are some constraints for arguments, in case of non-compliance which can cause some errors. And also problems can appear because of the memory problems with buffers.

2.3.1. Wrong arguments

If the format of the arguments does not meet the requirements, the program will either terminate, or print an appropriate error message.

Note: Sometimes it can show nothing, but just terminate. In these cases, we advise you to check the exit code, printing `echo $?` in terminal. If it's 1, then there's definitely some problem.

2.3.1.1) Not all options

For example, if we type:

```
./detector -i 10000 -t "%H:%M:%S" -c ls -la
```

which means no limit with its required option is given, the program will still run although the limit for launch is not provided. The point is that, it will run infinitely. In fact it's not an error, but the case where not all options are provided. So there are some option arguments that have values by default.

2.3.1.2) Wrong options

Also if we will try to execute program with wrong argument, for example

```
./detector -e 3 -i 10000 -t "%H:%M:%S" -c ls -l
```

it will print this error message: **invalid option - 'e'**

2.3.2. Memory allocation failure

If enough memory cannot be allocated while creating the buffer, program will exit.

2.3.3. Fork error

In case if there will appear problem while forking the process into the two processes using fork, the program will exit.

2.4. Implementation of test cases

The program should conform all 6 test cases, which were given in Moodle. In order to test if these tests are working or not, we have to execute each of them by typing **./test-(numberOfTestCase).sh**. If during the execution nothing will appear, it means that the program passed that test case. You can easily verify it also by typing **echo \$?** and see the successful return code which is 0 in case of success or 1 in case of failure. But this is very manual to run each test case individually. Instead the user can use one of the targets as `test`, `test-sans-valgrind` or `test-avec-valgrind` which will perform exactly the same thing. To use one of these targets just type *make targetname* in terminal. After these tests you can check the efficiency of your program typing *make gcov* in terminal which generates the code coverage report. Note that instead of running *test* and *gcov* targets separately, you can run just one target *couverture-et-tests* which is a shortcut for those two.

3. Contact us

Khaliq Aghakarimov - xaliq2299@mail.ru
Hikmat Pirmammadov - hikmet.pirmamedov@mail.ru

Table of contents

1. INTRODUCTION	1
1.1) What does this program do?	1
1.2) About the files of the project	1
2. USER MANUAL AND PROGRAM DESCRIPTION	2
2.1) Manual	2
2.1.1) Compilation	2
2.1.2) Usage	2
2.1.2.1) Syntax	2
2.1.2.2) Arguments	2
2.2) How does the program work?	2
2.2.1) Expected output	2
2.2.2) Examples of successful result	3
2.3) Possible errors	3
2.3.1) Wrong arguments	3
2.3.1.1) Not all options	3
2.3.1.2) Wrong options	3
2.3.2) Memory allocation failure	4
2.3.3) Fork error	4
2.4) Implementation of test cases	4
3. EXTRAS	4
3.1) Contact us	4
Table of contents	5