FIT2100 Lab01

Name : Wong Jun Kang
Student ID : 29801036
Tutor : Dr Ting Tin Tin
Lab Group : Wednesday 8am-11am

3.1 Task 1 (4 marks)

3.1.1 Task 1.1 (1 mark)

List all the command lines required for creating the following file hierarchy.
You can start creating the FIT2100 folder first.
(i.e. student@fit-vm:~/Desktop$ mkdir FIT2100)

cd Desktop
mkdir FIT2100; cd FIT2100
mkdir TUTORIALS
mkdir LABS; cd LABS
touch LAB1_Task1_1.txt

3.1.2 Task 1.2 (3 marks)

Experiment with navigating to the following paths using the cd command. After navigating to each directory, use the ls command to look at the files in that location. Are you able to navigate to all of these paths? Does it matter what working directory you are in before running the cd command?

- /home/student/Documents
- FIT2100
- ~/Documents/FIT2100
- .
- ..
- ../../../home/student
- /

(a) Which of the paths above are absolute, and which of them are relative paths

Path	**Absolute or Relative**
/home/student/Documents	Absolute
FIT2100	Relative
~/Documents/FIT2100	Relative

.	Relative
..	Relative
../../../home/student	Relative
/	Absolute

(b) What are the advantages of using absolute paths over relative paths? Why and when, if ever, would you choose to use relative paths over absolute paths?

i) Via an absolute path, we will be able to access the location of the file anywhere on the device as long as the file we refer to, stays and remains at the same location from the root directory.

ii) Relative paths are more location independent and hence are more “portable”. We should use relative paths when we are delivering/moving documents, toolboxes or data to another user or to another location of our computer, relative paths should be used. Otherwise, the recipient’s computer must have the same directory structure as ours. For example, when we have a program that makes use of a configuration file that is stored alongside the program executable, a relative path would be preferred to be used to refer to the configuration file. This is because when the location of entire program file is changed the location of the configuration file changes together (changes in the location of the file we refers to) and since a relative path is used, the configuration file will still be at the same relative location and can be located by the program, however, it cannot be accessed via an absolute path unless the user updates the file path explicitly.

(c) Write cd commands that allow you to achieve the following:

- navigate to one directory above the user's home directory
- no change to your current working directory

1. cd ~/..
2. cd .

(d) To run programs in your current directory, you will need to place a ./ in front of the name of a program. Why is this required? Assuming you have an executable named gcc in your current directory, describe what might happen if we were able to run programs in the current directory without needing to prepend a ./ to the program name.

The symbol “./” implies the current working directory. Hence, whenever we run a program we will first put a “./” to locate the current directory (as current directory is not in $PATH) and then run the program. If we are able to run programs in the current directory without needing to prepend a ./ to the program suggests that the program is located in the $PATH and hence we don't need to locate the current directory.

3.2 Task 2 (4 marks)

Enter the following program into a le named task2.c. Using gcc, compile two slightly different executables from your task2.c le. The rst executable, named task2, should be compiled as normal. The second executable, named task2-with-g, should be compiled using the -g option from the table in section 2.7.2 above.

```c
/* task2.c */
#include <stdio.h>

int main() {
    char string = 'N/A'; // Initialise our string

    printf("Enter a word, up to 10 characters long: ");
    scanf("%s", &string);

    printf("You entered %s\n", &string);
}
```

Then, answer the following questions (in bold text):

(a) Compare the file size of the two executables generated (try ls -lh to see the le sizes in `human' format). Why is one file larger than the other?

File size with -o option (Normal)	17KB
File size with -g option	19KB

File size with -o option (Normal) is 2 KB smaller compared to file size of executable created by -g option. This is because the c program that we compiled has errors and the file generated by option -g embed debugging information inside the executable file while file generated by -o command does not contain any debugging information which results in the difference in file size.

(b) You may have seen one or more warning messages. With good reason! While gcc is able to compile your code, the code is unlikely to run correctly, and may randomly produce a segmentation fault due to trying to access memory in an incorrect way.

Reading the warning messages, identify the line number and column number of the place in your source code that triggered the warning. (You do not need to x the code at this stage.)

Error message

```
student@fit-vm:~/Desktop$ gcc -g task2.c -o task2-with-g
task2.c: In function ‘main’:
task2.c:5:16: warning: multi-character character constant [-Wmultichar]
    5 |  char string = 'N/A'; // initialise our string
      |                ^~~~~
task2.c:5:16: warning: overflow in conversion from ‘int’ to ‘char’ changes value
 from ‘5123905’ to ‘65’ [-Woverflow]
```

c:5:16

Line number:5, Column number: 16

(c) Run your task2 executable through valgrind. Be patient, valgrind is much slower than running your program natively! Interact with your program and also read the extra information valgrind spits out. Is the information useful? Now try running your task2-with-g file instead.

```
student@fit-vm:~/Desktop$ valgrind ./task2
==8119== Memcheck, a memory error detector
==8119== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8119== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8119== Command: ./task2
==8119==
Enter a word, up to 10 characters long: ABABABABAB
You entered ABABABABAB
*** stack smashing detected ***: terminated
==8119==
==8119== Process terminating with default action of signal 6 (SIGABRT)
==8119==    at 0x48A318B: raise (raise.c:51)
==8119==    by 0x4882858: abort (abort.c:79)
==8119==    by 0x48ED3ED: __libc_message (libc_fatal.c:155)
==8119==    by 0x498F9B9: __fortify_fail (fortify_fail.c:26)
==8119==    by 0x498F985: __stack_chk_fail (stack_chk_fail.c:24)
==8119==    by 0x109201: main (in /home/student/Desktop/task2)
==8119==
==8119== HEAP SUMMARY:
==8119==     in use at exit: 0 bytes in 0 blocks
==8119==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==8119==
==8119== All heap blocks were freed -- no leaks are possible
==8119==
==8119== For lists of detected and suppressed errors, rerun with: -s
==8119== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Aborted (core dumped)
```

```
student@fit-vm:~/Desktop$ valgrind ./task2-with-g
==8441== Memcheck, a memory error detector
==8441== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8441== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8441== Command: ./task2-with-g
==8441==
Enter a word, up to 10 characters long: ABABABABAB
You entered ABABABABAB
*** stack smashing detected ***: terminated
==8441==
==8441== Process terminating with default action of signal 6 (SIGABRT)
==8441==    at 0x48A318B: raise (raise.c:51)
==8441==    by 0x4882858: abort (abort.c:79)
==8441==    by 0x48ED3ED: __libc_message (libc_fatal.c:155)
==8441==    by 0x498F9B9: __fortify_fail (fortify_fail.c:26)
==8441==    by 0x498F985: __stack_chk_fail (stack_chk_fail.c:24)
==8441==    by 0x109201: main (task2.c:13)
==8441==
==8441== HEAP SUMMARY:
==8441==     in use at exit: 0 bytes in 0 blocks
==8441==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==8441==
==8441== All heap blocks were freed -- no leaks are possible
==8441==
==8441== For lists of detected and suppressed errors, rerun with: -s
==8441== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Aborted (core dumped)
```

```
student@fit-vm:~/Desktop$ valgrind gcc -o task2 task2.c
==6698== Memcheck, a memory error detector
==6698== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6698== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6698== Command: gcc -o task2 task2.c
==6698==
task2.c: In function 'main':
task2.c:6:15: warning: multi-character character constant [-Wmultichar]
    6 | char string = 'N/A'; // inttialise our string
      |               ^~~~~
task2.c:6:15: warning: overflow in conversion from 'int' to 'char' changes value from '5123905' to '65' [-Woverflow]
==6698==
==6698== HEAP SUMMARY:
==6698==     in use at exit: 190,997 bytes in 107 blocks
==6698==   total heap usage: 415 allocs, 308 frees, 247,999 bytes allocated
==6698==
==6698== LEAK SUMMARY:
==6698==    definitely lost: 5,787 bytes in 36 blocks
==6698==    indirectly lost: 8,546 bytes in 8 blocks
==6698==      possibly lost: 0 bytes in 0 blocks
==6698==    still reachable: 176,664 bytes in 63 blocks
==6698==         suppressed: 0 bytes in 0 blocks
==6698== Rerun with --leak-check=full to see details of leaked memory
==6698==
==6698== For lists of detected and suppressed errors, rerun with: -s
==6698== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
student@fit-vm:~/Desktop$ valgrind gcc -g task2.c -o task2-with-g
==8017== Memcheck, a memory error detector
==8017== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8017== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8017== Command: gcc -g task2.c -o task2-with-g
==8017==
task2.c: In function 'main':
task2.c:6:15: warning: multi-character character constant [-Wmultichar]
    6 | char string = 'N/A'; // inttialise our string
      |               ^~~~~
task2.c:6:15: warning: overflow in conversion from 'int' to 'char' changes value from '5123905' to '65' [-Woverflow]
==8017==
==8017== HEAP SUMMARY:
==8017==     in use at exit: 191,021 bytes in 107 blocks
==8017==   total heap usage: 415 allocs, 308 frees, 248,279 bytes allocated
==8017==
==8017== LEAK SUMMARY:
==8017==    definitely lost: 5,787 bytes in 36 blocks
==8017==    indirectly lost: 8,546 bytes in 8 blocks
==8017==      possibly lost: 0 bytes in 0 blocks
==8017==    still reachable: 176,688 bytes in 63 blocks
==8017==         suppressed: 0 bytes in 0 blocks
==8017== Rerun with --leak-check=full to see details of leaked memory
==8017==
==8017== For lists of detected and suppressed errors, rerun with: -s
==8017== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Yes, valgrind spits out information regarding HEAP summary and memory leaks. It helps us in identifying overflows in memory (stack smashing detected) when we pass ‘N/A’ into a char type variable.

What is the difference in valgrind's output between the two different versions?

The only observable difference in the valgrind’s output between the two versions is the main showing the absolute path for file “task2” but the “task2-with-g”’s main is showing the c source file’s name with the total number of lines (i.e. line number of last line of code) in the c file.

(d) What is the program supposed to do? What is the program actually doing? Hint: Warning messages are a good way to understand what the program is doing wrongly. You might want to compare the warnings from the compiler against the output from valgrind.

The program was supposed to prompt the user to enter some characters (under length 10) and print “you entered <user input> characters”.The program first assigned ‘N/A’ (detected as int) to the string type variable, however, a single character was expected by the program’s string variable (char string) and ‘’ (single quotation), and since ‘N/A’ (more than 1 character inside a single. The number ‘5123905’ is the decimal representation of ‘N/A’. The binary representation of ‘5123905’ is “01001110 00101111 01000001” which is 3 bytes large, and char (‘’) is only 1 byte large, hence it overflows.

3.3 Task 3 (Extension: 0 marks)

Fix the program from Task 2, so that it does what it claims to the user it can do.

```c
/* task2.c */
#include <stdio.h>
#define SIZE 10


int main(){
    char string[SIZE]; // initialise our string
    
    printf("Enter a word, up to 10 characters long: ");
    scanf("%s", string);

    printf("You entered %s\n", string);
}
```