System Programming
2nd Laboratory (25th .. 28th February 2019)

C programming revision (debug and valgrind)

Summary

- Pipes and filters
- C programming revision
- Debug running processes
- Core dump and postmortem process analysis
- Overruns and leaks
- Uninitialized variables

In this laboratories all programs should be compiled with the options **-g -O0** (these options insert debug information into the programs and disable any optimization).
Although some of the errors are trivial and simple to correct, execute the programs with the error and just correct them after seeing the possible effects.

Pypes and filters
I

Using the pipes and filters architecture and several linux commands, count the number of processes the a certain user is running in the linux computer.
In Unix/Linux most utility programs can be used as filters, and pipes (the connection between filters) is represented by the character |
To accomplish this objective used the following utility programs:

- **wc**
- **grep**
- **ps**

Use the **man** command to understand what each program does and what should be the correct arguments.

Pipes and filters:

- http://homepages.cs.ncl.ac.uk/matthew.forshaw/teaching/csc8622/shell/04-pipefilter/
- Pipes and Filters Redux Robert Hanmer

Attaching processes to debugger
II

The program **infinite-loop.c** does not terminate. In order to understand where the infinite loop is, it is necessary to use the debugger.

I.a)

The easiest way is to start the program inside the debugger:

- **ddd inifinite-loop**
- press the button **run**
- after some time press the button **interrup**
- in the lower window observer the location
- in the lower window issue the command **where**

At this point it is possible to observer the point where the program has stoped.
If necessary issues the command **up** (several times) in the lower window.
When the debugger is in the upper frame (0x00000000004005af in main () at infinite-loop.c:10) it is possible to continue the program step by step:

- issue the command **display i**
- press **step** several times
- observe the value of **i**

I.b)

If the programm is running in the shell it is necessary to attach the debugger to the running process:

- start the program **inifnite-loop** in a terminal
- start a new terminal
- in the new terminal
  - issue the command **ps -a | grep infilite-loop**
  - take note of the process ID (first number on the output of the command)
    - for example ***658***
  - issue the command **ddd inifinite-loop**
  - in the lower window of **ddd**
    - issue the command **attach 658** (it may be other value)

From this moment on, it is possible to use the debugger as if the application was started inside the debugger: (**where**, **up**, **print**, **display**, …)

I.c)

Correct the program.

DDD manual:
https://www.gnu.org/software/ddd/manual/

Core dump
III

Sometimes the program crashes dues to invalid pointers. When this happens the program stops running, a message is printed in the terminal and if configured a core file is generated:

- compile the program char-conv.c
- execute it in the terminal
- write a word and press enter

II.a)
If the program is executed inside the debugger (**ddd**) from the beginning, when the programs crashes, the debug signals the crash location and allows the programmer to see the incorrect values:

- in the terminal run ddd: **ddd char-conv**
- inside ddd
    - run the program (press the button **run**)
    - write a word in the lower window
    - wait for the program to crash (a red arrow will show the wrong line)
    - issue the command **where**
    - issue the command **print v1**
    - issue the command **print v2**

The address of v2 is invalid!
Exit ddd.

II.b)
In order to do a postmortem evaluation of the program that was executed in the terminal, it is necessary for the operating system to generate a core dump file.
These command work on ubuntu, may not work on other versions of linux :(
In the terminal:

- issue the command **ulimit -c unlimited**
- execute the application (**char-conv**)
- write a word
- the program crashes and a **core** file is generated
- issue the command **ls**
- execute the debugger issuing the command **ddd char-conv core**
- the debugger presents the location of the crash

Correct the error.

Overruns and leaks
IV

The directory **III** contains a possible correction to the previous exercise (**char-conv-prob.c**). Although working correctly this program has two programming errors.
In order to find these issues **valgrind** can be used:

- compile the program
- run the program inside **valgrid**
  - issue the command: **valgrind --leak-check=full -v ./char-conv-prob**
- observe the output

The last message of valgrind states that the program has 2 errors:

```
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

III.a)
Memory leak
The first error is a memory leak:

```
==2985== HEAP SUMMARY:
==2985==     in use at exit: 7 bytes in 1 blocks
==2985==   total heap usage: 3 allocs, 2 frees, 2,055 bytes allocated
```

this message informs that a malloc was made but no free was performed before the end of the program.

Correct this error.

III.b)
Buffer overruns
Valgrind also identifies a **Invalid read of size 1 ….**
This error means that during the program execution a read operations tried to access a memory outside a valid array:

- Address 0x51db8c7 is 0 bytes after a block of size 7 alloc'd

Valgrind also indicates that the memory was allocated in **main (char-conv-correct.c:15)** and the access was performed in **main (char-conv-correct.c:20)**

Correct this error.

Valgrind memcheck manual:
    http://valgrind.org/docs/manual/mc-manual.html

Uninitialized memory
V

The directory **IV** contains a possible correction to the previous exercise (**char-conv-uninit.c**). Although working correctly this program still has one programming error.
In order to find this issues **valgrind** can be used:

- compile the program
- run the program inside valgrid
- issue the command: **valgrind --leak-check=full -v ./ char-conv-prob**
- observe the output

Valgrind states that a **Conditional jump or move depends on uninitialised value(s)** on line **main (char-conv-uninit.c:20)**. This happens because the conversion loop did not copy the '\0'

Correct the error.

Apps vs servers
VI

In the previous exercises (III and IV) we identifies three different memory allocation related errors. Describe why the programs managed to run correctly even in the presence of these errors:

For each of the previous errors describe how their presence in a server running continuously can affect the availability of the server:

Memory leak	
Buffer overrun	
Uninitialized memory	