Exercises lecture 3

# Userspace vs Kernelspace

In this exercise we will write, compile and run our first and second C program. First we will create a typical “hello-world”-file. Afterwards we will create a **Kernel module**. Refer to the theory about the difference between writing programs that run in user space and programs or processes that run in kernel space. **Do the following exercise in a virtual machine (Linux, Debian) and create a snapshot to revert to a previous state if needed!!**

* 1. Install a c compiler(gcc), a building tool (make) and your favorite editor:   
     sudo apt install gcc gdb g++ make strace vim nano -y
  2. Let’s search for the correct linux headers of our system:
     1. apt search linux-headers-$(uname -r)
     2. The previous command should give you a package name: something that looks like “*linux-headers-4.19.0-10-amd64*”
     3. Install the linux headers (change numbers if needed):

sudo apt install linux-headers-4.19.0-10-amd64 -y

* 1. Let’s try to create our helloworld program in a separate folder.
     1. Create a new folder: *mkdir helloworld*
     2. Go in the folder helloworld: *cd helloworld*
     3. Create a new file hello.c: *nano hello.c* (of vim hello.c)
     4. Put the following contents in that file



* + 1. On the cli type:

make hello

* + 1. It should output something like “cc hello.c -o hello”
    2. Look at the contents of your current directory (*ls*). A new file was created. Wat is the extension?
    3. Try to cat the contents of this new file. What do you see?
    4. Try to use the *file* command to know what type of file this is (file hello)?
    5. Run the program by typing ./hello
    6. You should see “Hello World!”
  1. Let’s create a testcrash.c file in a separate folder.
     1. Go back one level (cd ..)
     2. Create a new folder: mkdir testcrash
     3. Go in the folder: cd testcrash
     4. Create a new file: nano (or vim) testcrash.c
     5. Put the following code:



Do once again a “make testcrash” and a “./testcrash”

**What happens when you run it?**

In the testcrash.c file we did something that is not allowed. Refer to the theory why this is an invalid operation. No problem though as this program ran in userspace. Now we will try to do a similar thing but in a kernel module. In other words we will create a program that will run in kernel space!

1. Create 2 folders “spyro” and “bandicoot” – mkdir bandicoot
2. In these folders create a “Makefile” with no extension. In the spyro folder create a “spyro.c” and in the bandicoot folder create “bandicoot.c”.
3. Paste the following contents in the Makefile. Note the tab before make! This might cause issues when copying. Replace the bandicoot.o to spyro.o for the other makefile.



1. Paste the following contents in “spyro.c”



1. On the cli type: make
2. Open a new shell or tty (through ssh or with “alt and f2”) and type:   
   (sudo or as root) dmesg -w
3. Next we will insert/activate the kernel module in the kernel. As root with (su -) or with sudo: insmod spyro.ko
4. To be sure you can check all modules by issuing the lsmod command. Try to grep on spyro. Is it in the list? Now remove the module with rmmod spyro. Try to grep again. It should be gone. Take a look at the /var/log/messages. If everything went right you should see our info messages.
5. Congratz!



1. Let’s continue with bandicoot.c. Put the following code in the file.



1. Once again, insert the module and try to remove it. You will see it won’t work, (why?). Now take a look at the /var/log/messages. What do you see?
2. In the directory with bandicoot.c type “make clean”. All generated files should disappear. Copy bandicoot.c to crashbandicoot.c and replace bandicoot to crashbandicoot in the Makefile.
3. Now paste the following in crashbandicoot.c and do the same (be careful to only run this in a virtual machine that has a snapshot ;)



1. Explain why we get a freeze and why we got a segmentation fault previously.

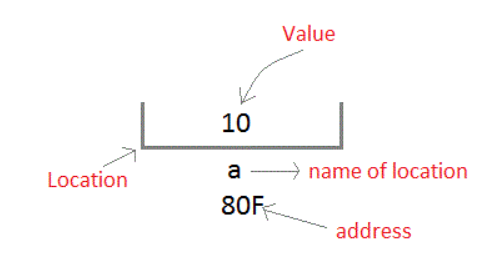
# Pointers & Memory addresses in c

Whenever a variable is defined in C, a memory location is assigned to it. At this memory location the value of the variable is stored. You can check this memory address with the “&” symbol.

Go ahead and try to run the following code by creating a “var.c”-file and running “make var” or “gcc var.c -o var”. Afterwards run it by typing “./var” on the commandline.



This means that whenever we declare a variable in our program, the system allocates an address in memory. This location has its own address number, which we saw in the previous example. Imagine we have a line “int a = 10;”. A visual representation of that statement would be something like this:

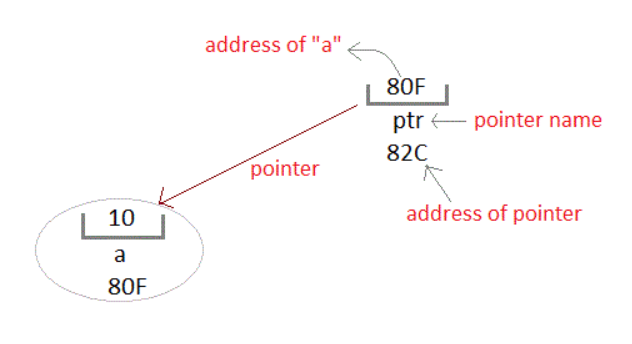


To retrieve (or access) the value 10, we could use the name of the variable (a). Another way to get the value is telling the system to look at the exact location (address). Imagine having your smartphone laying on a table. You could ask “could you hand me over my smartphone” or “could you hand me over what is on the table”. In the latter you hope the table is not something like this:



This is exactly the reason why it is important to only use the memory you need for the stuff you put there! No need to have 1.000.000.000 bytes to store one integer value.

So how do we access a variable using it’s address? The answer is **pointers!** We will create a new variable that has as value the address of our variable.

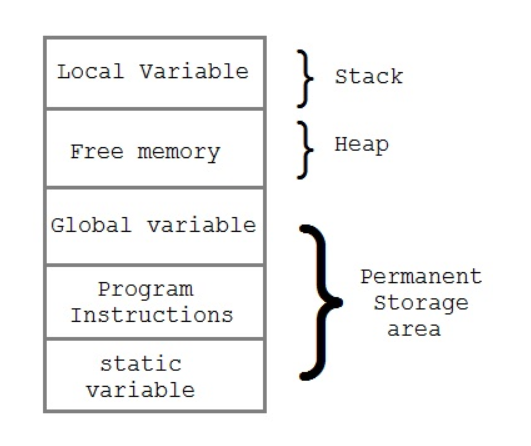


Now try to predict the output of the following program and afterwards run it to check your answer. After running, try to answer the “what happens here”- questions.



Up until now we only created variables that store int’s. What about objects? Well it is actually the same. You have objects on a memory location and it is possible to have pointers “pointing” at the address locations of the objects. There is only one small problem when talking about objects like you know them from java. C does not have classes, it does have structs… don’t worry about the details.

Global variables, static variables and program instructions get their memory in a permanent storage area whereas local variables are stored in a memory area called the “Stack”. The memory space between these two region is known as the “Heap” area. This region is used for dynamic memory allocation during execution of the program. The programmer is responsible for allocating and freeing this memory!



The following program illustrates how one can allocate memory dynamically with the help of malloc. You can always take a look at the man-page of malloc. C does not understand strings by design. But it does understand characters. Luckily for us multiple characters form a string! Imagine having a vector/array/table (= these are all the same in this context) of char’s. The string “hello” is the same as a collection (array) with h on index 0, e on index 1,…

Note: we need something to finish the sequence! Something to tell “this is the end of my string”. This is is a “\0”.



Run the program and try some things out:

* Run it normally (no unexpected behavior).
* Run it where your limit is 1 and you put a really long sequence of characters.
* Run it and try to put text as limit.

Try to explain (with a drawing) how this program works. How is it possible that we get a string by only using a pointer to a char? Can you also explain why we need to put enter a limit? **NOTE: Modern systems allocate enough memory for a process & modern operating systems are able to bypass “trivial mistakes”. Chances are that you will be able to use the program without having any issues.**

Finally, why do you think it is important to free your memory with free()? Try to remove (or put the free in comments) and recompile. Now use the tool valgrind (apt install valgrind) on your program.

“valgrind --leak-check=full ./var”

And enter “2” as limit and “a” as input. You should see more “allocs” than “frees”.

Recompile again with a correct free and the same limit and input. What does the tool tell you now? Finally try to run valgrind when you try to put too many characters.

If you are interested in more C you can look at <https://www.studytonight.com/c/overview-of-c.php> for a brief overview, most images and the above exercises found their origin there. Do remember that C or C++ is not material for the exam!

For those who are interested in C++, instead of using malloc they introduce the keyword “new”. This should sound familiar from Java ;) Modern C++ (from C++11) even introduced the concept of smart pointers. By using this mechanism, you don’t have to remember to free your memory! A lot safer and less error prone.