ÓBUDA UNIVERSITY
JOHN VON NEUMANN FACULTY OF INFORMATICS

Linux kernel and C programming

Author: Gergely Korcsák
Date of issue: February 9, 2025

Contents

1 Introduction

This paper is for the introduction of the C programming language, and its usage in the linux kernel. Reading this you will also learn how to create linux kernel modules, and communicate between them and user applications.
Please not that, altaugh this is ment to be an introductory paper and course, this paper will assume that you have previous programming experience, and know how to naviagate the linux filesystem. Therefore it will not contain full explanation for basic linux commands and howtos, only examples. Basic environment setup will be provided.

1.1 Prerequirerities

First we will need a linux machine, which can be obtained trough many ways. You can eather use a Virtual Box, your own machine, or WSL. *It is advised to use a VM leter on the course, as crashing the kernel when programming modules is common.*
Lets see how to enable WSL on Windows:

```
1 PS > wsl --install                    # Enable the windows linux subsystem
2 PS > wsl --list --online              # List all available VMs
3 PS > wsl --install -d Ubuntu          # Install an Ubuntu virtual machine
```

Code Snippet 1.1: Installing WSL

From here on out, the following commands will assume, that you are using Debian/Ubuntu. If that is not the case, *I assume you have the proper knowledge of how to use your own system/distribution*.

We will need to following programs trough the course:

- gcc or clang
- make and kernel-make
- cmake

To install them, run:

```
1 $ sudo apt install gcc clang make cmake build-essential kmod
```

Code Snippet 1.2: Installing WSL

2 The C Programming language

2.1 History

The C programming language is very simple. Altaugh using the language can be quite difficult, due to handling memory directly, and the easyness of shooting yourself into the foot. But at the same time, it also provides the freedom to use the hardver exactly how we *written* it to be used.

C is a general-purpose programming language, and was created in the 1970s by *Dennis Ritchie*. It is widely used and most of our infrastructure runs on it.
It was mainly developed to replace assembly, and create portable code/programs. Altaugh it is somewhat achived, but moving our code from one platform to another is only available with the help of general system headers.
Programs written in C will run at the bare metal, and doesn't have any garbage collection like modern high level languages.
Some competitors and alternatives are: `zig`, `rust` and `go`. They provide memory management, and compile time memory checks, which will be mandatory in the future, defined by the US government.

2.2 Types

C is a strongly typed programming language. To declare a variable you need to write the `type` before it. The size of a variable will depend on the architecture, but lets see the sizes for a 64 bit processor.

Type	**Size**	**Description**
`char`	`8 bit`	Signed Character (treated as int)
`short`	`16 bit`	Short Signed integer.
`int`	`32 bit`	Signed integer.
`long`	`64 bit`	Long Signed integer.
`float`	`32 bit`	Single precision floating point.
`double`	`64 bit`	Double precision floating point.

Table 2.1: Basic C Types

You can read out the bytesize of a variable by using `sizeof()`.
We can use the following modifyers to change the size and signage of a type.

Modifier	**Description**
`signed`	Ensure a variable is signed.
`unsigned`	Ensure a variable is unsigned.
`short`	Shortens the variable size by half. *(if possible)*
`long`	Doubles the variable size. *(if possible)*

Table 2.2: Type modifiers

2.3 Variables

Variables are not zeroed out at initialization. Therefore when we declare a variable, by default they will have the previous value of the memory they were assigned to.

```
int a;
int b = 0;

int c = 2, d = 4;
```

Code Snippet 2.3: Creating variables

We can do math between them, by using the `+`, `-`, `*` and `/` operators.

```
a = d - b;       // a: 4
int e = a * c    // e: 8
```

Code Snippet 2.4: Using basic math

2.4 Macros

Macros can be interpreted as compile time variables. The can be defined by `#define name value` C doesn't have boolean types, so they can be defined as:

```
#define false   0
#define true    1
```

Code Snippet 2.5: Macro example bools

Macros will only be used at preprocessing, and will be inserted into the code as is. They can contain function like variables, which will be used at preprocessing. Lets se a short example:

```
#define debug_string    "[debug]: "
#define mystring        "words in my head"
#define number1         75434
#define number2         1234
#define sum(a, b)       a + b
#define mul(a, b)       (a * b)

printf(debug_string "\%s - \%d\n", mystring, sum(number1, number2));

// will be compiled as:
printf("[debug]: " "\%s - \%d\n", "words in my head", 75434 + 1234));
// will print: [debug]: words in my head - 76668
```

Code Snippet 2.6: Macro examples

In this example we can also see that constant strings will be concatenated if they follow each other. So `"[debug]: "` and `"%s - %d\n"` will be treated as one string by the compiler, as: `"[debug]: %s - %d\n"`.

2.5 Printf

`printf()` is part of the standard library. It requires a format string, and extra arguments if they are defined in the format string.
The syntax is:

```
#include <stdio.h>
printf("format_string", args...);
```

Code Snippet 2.7: printf format string

The format string can contain format specifiers:

Specifier	Description
%d	Signed integer
%u	Unsigned integer
%c	Character
%f	Floating-point number
%x	Hex number format
%s	String closed by a zero character \0
%%	Print the % character

Table 2.3: printf format specifiers

The format specifier can also be extended for prefilling by writing the charecter and the length of characters printed. For example:

```
printf("My Hex: 0x\%08x\n", 0xAB13);         // will print: My Hex: 0x0000ab13
printf("My Hex: 0x\%08X\n", 0xAB13);         // will print: My Hex: 0x0000AB13
```

Code Snippet 2.8: printf examples

The precision of floating-point numbers can also be specified as:

```
printf("My float: \%.2f\n",      2.3456);    // will print: My float: 2.34
printf("My float: \%02.1f\n",    2.3456);    // will print: My float: 02.3
```

Code Snippet 2.9: printf float formatting

2.6 Basic memory and pointers

You can look at computer memory as an array of bytes. It can be indexed from 0 to the sky. When running your program you will get a slice of memory to use from the kernel. This is where your application will be loaded.
When checking the address of a variable inside your program, you will see that, your program isn't running on address 0. That address is reserved for the bootloader. Furthermore if you check the address of the function `main`, you will also see, that it is not even near of the address of a variable inside the program. We will talk about that further in section 2.17.
For now, you can think of the memory as one addressed block of storage.

2.6.1 Pointers

Pointers are also variables, which point to an arbitrary memory location. Their value can be read as an integer and also be used to modify a value at a specific location. Lets create a variable and a pointer to its address:

```
1 int a = 10;
2 int* p = &a;     // will set the value of 'p' to the address of 'a'
3 *p = 5;          /* will set the value at the address inside 'p' to 5,
4                     therefore 'a' will be 5 */
5 p = 10;          /* will set the value of 'p' to 10,
6                     the value of 'a' will stay 5 */
```

Code Snippet 2.10: Pointers

2.6.2 Pointer arithmetics

Modifier	**Description**
`&p`	Get the memory address of variable `'p'`.
`*p`	Get the value pointed by `'p'`.
`p++`	Multiply the value of `'p'` by the bytesize of its type. If `'p'` is an `(int*)`, then it will be multiplied by `(4)`. If `'p'` is a `(char*)`, then it will be multiplied by `(1)`.
`p--`	Substract the value of `'p'` by the bytesize of its type. If `'p'` is an `(int*)`, then it will be substracted by `(4)`. If `'p'` is a `(char*)`, then it will be substracted by `(1)`.
`p + n`	Multiply the value of `'p'` by the bytesize of its type times `n`. If `'p'` is an `(int*)`, then it will be multiplied by `(4 * n)`. If `'p'` is a `(char*)`, then it will be multiplied by `(1 * n)`.
`p - n`	Substract the value of `'p'` by the bytesize of its type times `n`. If `'p'` is an `(int*)`, then it will be substracted by `(4 * n)`. If `'p'` is a `(char*)`, then it will be substracted by `(1 * n)`.
`p[n]`	The same as `*(p + n)`. We are multiplying the value of `'p'` by the bytesize of its type times `n`, then getting the value at that address. Lets continue in section 2.7.

Table 2.4: Pointer arithmetics

2.7 Arrays

In C, arrays are pointers, with a predefined length of memory. They can be declared like the following:

```
int array[10];
int array[5] = { 0, 1, 2, 3, 4 };
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Code Snippet 2.11: Array declaration

We can access the element of an array, by its index like:

```
array[0] = 2;
array[2] = 4;
int a = array[3];
```

Code Snippet 2.12: Array use

We can also user pointers as arrays:

```
int array[10];
int* p = array;
```

Code Snippet 2.13: Pointer of an array

An array is just a pointer to an already allocated memory area. Therefore we can also get the n-th value like the following:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int a = *(array + 3);   // a will be 3
```

Code Snippet 2.14: Array with pointer arithmetics

2.8 Conditions

C don't have booleans, therefore a condifion is true if it is not 0, and false if it is.

```
if (0) {
    // Will always be false
}
if (1) {
    // Will always be true
}
if (-5) {
    // Will always be true
}
```

Code Snippet 2.15: Conditions

We can use the following conditional operators:

Operator	Description
!	Not
>	Greater than
<	Lesser than
==	Equals
!=	Not equals
&&	And
||	Or

Table 2.5: Conditional operators

2.9 Strings

Strings are an array of characters closed by 0.

```
char* str = "Hello strings!";
```

Code Snippet 2.16: Strings

2.10 Loops

We can talk about `while`, `do while` and `for` loops.
While and do while will run until the provided condition is not 0.

```
int i = 5;
// Will print the numbers from 5 to 1
while (i) { // can also be (i > 0)
    printf("\%d\n", i);
    i--;
}

// Will print the numbers from 0 to 4
while (i < 5) {
    printf("\%d\n", i);
    i++;
}
```

Code Snippet 2.17: While loop

```
int i = 0;
// Will print the numbers from 0 to 4
for (i = 0; i < 5; i++) {
    printf("\%d\n", i);
}
```

Code Snippet 2.18: For loop

2.11 Bitwise operations

2.12 Functions

2.13 Struct and Typedef

2.14 Unions

2.15 Using headers

2.16 Static, Const and Extern

2.17 Program Memory

2.18 Pointers extended

3 Compilation

3.1 Hello World

Lets write our first application. Create a `hello_world.c` file, and write the following:

```c
#include <stdio.h>

int main() {

    printf("Hello World!\n");

    return 0;
}
```

Code Snippet 3.19: hello$_world.c$

Lets compile our program with:

```
gcc hello_world.c -o hello_world
```

Code Snippet 3.20: hello$_world.c$

List of Tables

Code snippets