Operating Systems: Deep C Dive

1 Introduction

In this lab we will look at some C basics by writing a number of programs. The relevant code is available at in the deep-c-dive directory.

2 Background:

Let's briefly discuss some of the concepts that will be used in the programs in this lab.

2.1 Pointers

In C, pointers are variables that store the memory address of another variable. Instead of holding a data value directly, a pointer holds the address where the data is stored in memory.

2.1.1 Function Pointers:

Function Pointers are pointers, i.e. variables, which point to an address of a function. Functions like variables, can be associated with an address in the memory. A specific function pointer variable can be defined as follows.

```
int (*fn)(int,int) ;
```

Here we define a function pointer fn, that can be initialized to any function that takes two integer arguments and return an integer.

A function can take many types of arguments including the address of another function. Thus, function pointer can also be passed as an argument to another function.

2.1.2 Pointer Arithmetic:

There are a set of valid arithmetic operations that can be performed on pointers. Pointers store memory addresses of another variable. So, increment/decrement and addition/subtraction operations can be done on a pointer variable. Increment/decrement of a pointer increments/decrements it by a number equal to the size of the data type for which it is a pointer. E.g., an integer pointer, if

incremented, will increment by the size of an integer. When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

2.2 Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The structure tag refers to an (optional) name for the structure.

The syntax to declare a structure is as follows-

```
struct [structure_tag]{
    member1 definition1;
    member2 definition2;
    ...
    member3 definition3;
} [structure_variables];
```

2.2.1 Initialization

A structure can be initialized by placing the value of each variable within curly braces, or using the `dot(.)` operator. They can also be initialized by designated initializers.

Designated Initializers:

```
struct structure_tag structure_variable = {.definition1 = value1, \
.definition2 = value2, .definition3 = value3};
```

Structures can be nested, i.e., we can have one struct inside another struct.

2.2.2 Padding

Structure padding is the addition of some empty bytes of memory in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.

2.3 C standards

C standards define the specifications for the C programming language. They describe the syntax, semantics, and behavior of the language, ensuring that C programs can be compiled and executed consistently across different platforms and compilers. Over the years, several versions of the C standard have been released, each introducing new features, fixing issues, and improving language support.

You can learn more about the current C standard here- International Standardization Working Group, Current C, What is C23

2.4 GCC Extensions to C

GNU C provides several language features not found in ISO standard C- gcc extensions.

2.4.1 Attributes

In C, an attribute is a special keyword or directive that provides additional information about the behavior or properties of a function, variable, or type. Attributes are used to give hints to the compiler about optimization, alignment, or other aspects of how the compiler should handle the given code.

Some commonly used attributes are listed below:

`deprecated`:

This attribute is used to mark a function as deprecated. When a deprecated function is called, it triggers a compiler warning to inform the programmer that this function is outdated and should not be used in the future.

`always_inline`:

This attribute forces the compiler to always inline the function. Inlining a function means that the function's code is inserted directly at the point of the call, potentially improving performance for small, frequently used functions.

`aligned`:

This attribute is used to align a variable in memory at a specified boundary. The program checks whether the variable is correctly aligned to by verifying its memory address.

`noreturn`:

This attribute marks a function as not returning. This tells the compiler that the function will not return control to the calling function, which can help optimize the code.

`unused`:

This attribute indicates that a function parameter is intentionally not used. It prevents warnings for unused variables or parameters, which can be useful when dealing with certain library functions or placeholders.

2.4.2 Labels as Values

In C programming, a label is an identifier followed by a colon (:) that marks a specific location in the code. Labels are used in conjunction with goto statements, which allow the program's execution to jump to a different part of the code.

Label as values language extension can be used to compute the address of a label using `&&` operation and compute `goto` by using the `*` indirection operator.

2.4.3 Zero length arrays

Zero length arrays are useful in certain scenarios like managing flexible memory allocation or creating structures with a variable-length component at the end.

2.5 C Memory Layout

The memory layout of a program refers to how the program's data is stored in the computer memory during its execution.

A C program's memory is organized into specific regions, each serving distinct purposes for program execution, as discussed in C Memory Layout.

2.5.1 Text Segment

The text segment (also known as code segment) is where the executable code of the program is stored. It contains the compiled machine code of the program's functions and instructions.

2.5.2 Data Segment

The data segment stores global and static variables that are created by the programmer. It is present just above the code segment of the program. It can be further divided into two parts:

- **Initialized Data Segment**: It is the part of the data segment that contains global and static variables that have been initialized by the programmer.
- **Uninitialized Data Segment (BSS)**: Uninitialized data segment contains global and static variables that are not initialized by the programmer. These variables are automatically initialized to zero at runtime by the operating system.

2.5.3 Heap Segment

Heap segment is where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows towards the larger addresses from there.

2.5.4 Stack Segment

The stack is a region of memory used for local variables and function call management. Each time a function is called, a stack frame is created to store local variables, function parameters, and return addresses. This stack frame is stored in this segment.

The stack segment is generally located in the higher addresses of the memory and grows opposite to heap. They adjoin each other so when stack and heap pointer meet, free memory of the program is said to be exhausted.

3 Programs

3.1 pointer_demo.c

The program demonstrates the usage of pointers in C, including pointer declaration, dereferencing, updating pointer references, and handling null pointers. It prints values and memory addresses using both direct variable access and pointers.

The program declares an integer variable `num` and initializing it with a value. A pointer `ptr` is then assigned the address of `num`. The program prints the value and address of `num` using both direct access and the pointer.

Next, the value of `num` is modified using the pointer, demonstrating how pointers allow indirect modification of variables. The pointer is then updated to point to another integer variable, `anotherNum`, illustrating how pointers can be reassigned.

Finally, the pointer is set to `NULL` to indicate it no longer references a valid memory location. However, the program attempts to dereference the null pointer, leading to undefined behavior.

3.1.1 Conclusion:

This program illustrates the fundamental operations on pointers in C, including dereferencing, pointer reassignment, and handling memory addresses. However, dereferencing a null pointer results in undefined behavior and should be avoided in practice.

3.2 fun_pointer.c

This program demonstrates the usage of function pointers in C, including their declaration, assignment, and passing them as arguments to other functions.

The program defines two arithmetic functions, `add` and `subtract`, which perform addition and subtraction, respectively. A function pointer `funcPtr` is dynamically declared and assigned to these functions, demonstrating how function pointers allow flexible execution of different functions at runtime.

Additionally, the program includes the `operate` function, which takes a function pointer as an argument. This illustrates how function pointers enable higher-order functions, allowing a function to operate on different functions dynamically.

The program prints results obtained from calling functions using pointers and passing function pointers as arguments. This program also tries to read the first byte by dereferencing the function pointer. This value will depend on the system and architecture but for linux x86 system, it might allow to read the first instruction of compiled machine code subtract() function.

3.2.1 Conclusion:

This program highlights the versatility of function pointers in C.

3.3 fun_pointer_address.c

Similar to `fun_pointer.c`, the program prints the memory address of the function pointer to illustrate how functions are stored in memory and accessed via pointers.

3.4 pointer_arithmetic.c

The program demonstrates pointer arithmetic in C, where the pointer is manipulated to traverse through an array and perform operations like incrementing, adding offsets, and calculating the distance between two pointers.

The program begins by defining an array numbers[] containing five integers. A pointer ptr is initialized to point to the first element of the array. The program then prints the address and value of the first element using the pointer.

The pointer is incremented (ptr++) to point to the next element in the array, the pointer is then moved forward by two elements using ptr += 2. Similar operations are done with decrementing the pointer as well. The pointer type determined the arithmetic by the pointer and points to different elements in the array. Addresses and values of the array element are printed.

Finally, the program calculates the distance between two pointers (startPtr and endPtr) that point to the first and last elements of the array, respectively. The distance is computed in terms of array elements, and the result is printed. Since they variables `startPtr` and `endPtr` are of integer pointers, the distance is calculated as array elements but if they were to be cast as character pointers, this distance would change to the number of bytes between them.

3.4.1 Conclusion:

This program illustrates the use of pointer arithmetic to navigate through an array, including pointer increment, addition of offsets, pointer decrement, and pointer subtraction. It also demonstrates how to compute the distance between two pointers in terms of the number of elements in the array.

3.5 struct.c

The program demonstrates the use of `structures` in C by defining and working with a `struct Student`. It showcases how to declare, initialize, and access structure members, as well as how to pass structures to functions.

The `struct Student` contains three members: an integer `id`, a character array `name` for storing the student's name, and a floating-point `grade`. The function `printStudent` takes a `Student` structure as an argument and prints its details.

In the `main` function, `Student` structures (`student1` and `student2`) are declared and initialized in two ways: by directly assigning values to its members and by using designated initializers.

3.5.1 Conclusion:

This program demonstrates how to define and use structures in C, including different ways to initialize structure members and pass structures to functions.

3.6 nested_struct.c

The program demonstrates the use of nested structures in C by defining and working with a `struct Person` that contains an embedded `struct Address`. It showcases how to declare, initialize, and access nested structure members, as well as how to pass structures to functions.

The nested structure can be accessed like a member of the struct and can be initialized by designated initizlizers.

3.6.1 Conclusion:

This program demonstrates how to define and use nested structures in C.

3.7 padding.c

The program demonstrates struct padding and memory layout in C by defining a simple structure `struct Example` and examining its memory size. It also explores pointer manipulation to highlight how memory is allocated for structure members.

The `struct Example` consists of an `int` (`i`) and a `char` (`c`). Given that an `int` typically requires 4 bytes and a `char` requires 1 byte, one might expect the structure to occupy 5 bytes. However, due to structure padding, the compiler aligns the structure to optimize memory access, potentially increasing its size.

In the `main` function, the program prints the sizes of `char`, `int`, and `struct Example` using `sizeof`. The actual size of `struct Example` is often 8 bytes due to padding added by the compiler for memory alignment.

The program then declares a structure variable `a` and assigns a character pointer `p` to its address. The line `p[6] = 1` attempts to modify memory at an offset of 6 bytes from the start of `a`, which might result in undefined behavior if the structure size is less than 7 bytes.

3.7.1 Conclusion:

This program highlights the concept of structure padding and memory alignment in C. The actual size of a structure may be larger than the sum of its individual members due to automatic padding by the compiler.

3.8 empty.c

This program makes use of the `volatile` keyword. `volatile` tells the compiler not to optimize anything that has to do with the volatile variable.

3.9 print_standard.c

This program prints the version of the C standard that the compiler is using.

3.10 c89.c

This program demonstrates the difference between different C standards. The _Noreturn function specifier in C was introduced in `C11`. The `c89` standard also used * *\ for comments instead of \\, and required explicit function arguments. So, this program will not compile in `c89` standard but it will compile in newer standard.

The _Noreturn function specifier allows the compiler to apply Tail Call Optimization in certain cases. A tail call is a function call that is the last operation before returning in a function. Tail Call Optimization (TCO) is a compiler optimization where instead of pushing a new stack frame for a function call, the current stack frame is reused. This avoids stack growth, preventing unnecessary memory usage and potential stack overflow.

In this program, since `fun()` does not return (infinite loop) and does not need further execution in `main()` (the next statement is never reached), the compiler can optimize away the stack frame for `main()` and directly jump to `fun()`, reusing the same stack frame instead of creating a new one.

3.10.1 Conclusion:

This program highlights the differences in different C standards.

3.11 attributes.c

The program demonstrates various function and variable attributes in C, showcasing how they can be used to modify the behavior and optimizations in the code.

The program defines several functions with different attributes:

`deprecated:` This attribute is used to mark a function as deprecated. When `oldFunction` is called, it triggers a compiler warning to inform the programmer that this function is outdated and should not be used in the future.

`always_inline`: This attribute forces the compiler to always inline the function `inlineFunction`.

`aligned(16)`: The program checks whether the variable `alignedVar` is correctly aligned to 16 bytes by verifying its memory address.

`noreturn`: This attribute marks `exitWithError()`, as not returning. This tells the compiler that the function will not return control to the calling function (because it calls `_exit()`), which can help optimize the code.

`unused`: This attribute indicates that a function parameter, such as `x` in `unusedParameterDemo()`, is intentionally not used.

3.11.1 Conclusion:

This program demonstrates how function and variable attributes can be used to influence compiler behavior and optimizations.

3.12 labels.c

The program demonstrates the use of labels as values using computed `goto`s in C, where labels are stored in an array using the address-of label operator (`&&`) and used for program control flow, allowing the program to "jump" to different parts of the code based on user input.

3.13 zla.c

The program demonstrates the use of a zero-length array within a structure. It defines a `DataPacket` struct with a zero-length array `payload[0]` to hold variable-length data. The main function allocates memory dynamically for the struct and the payload, then copies a message into the zero-length array, and finally prints the contents of the struct.

The zero-length array, `payload[0]`, is used as a flexible array member (FAM) in the struct. It allows the struct to be allocated with extra space to hold a variable-length payload. The program uses `malloc` to allocate memory for both the struct itself and the message that is copied into the payload.

In the program, the length of the message is computed using `strlen`, and `malloc` allocates memory for both the struct and the message. After assigning an ID and copying the message to the payload, the program prints the struct's contents. Finally, the memory allocated is freed using `free`.

3.13.1 Conclusion:

This program highlights the concept of zero-length arrays in C, specifically flexible array members (FAMs), which allow dynamic memory allocation for structures with variable-length data.

3.14 typeof.c

This program demonstrates the use of a macro to safely swap the values of two variables.

```
typeof(a) temp = a;
```

This line declares a temporary variable temp that has the same type as the variable a. This is done using `typeof`, which is a GCC extension to obtain the type of a variable or expression. Since the temporary variable is declared as such, it we can use this macro for both `int` and `float` without any issues.

3.15 global_static.c global_static_init.c global_init_static_init.c

Using the `Makefile`, (`make layout` command), we can make use of the `objdump` command to observe the behavior of a C program with respect to global and static variables, and the associated memory sections.

Uninitialized global variable `global_var` and uninitialized static local variable `static_var` are stored in uninitialized data segment of memory. `common` section is used for uninitialized global variables and uninitialized static local variables go in `_bss`. Initialized variables go to data segment (DS).

3.16 hello.c

Using the `Makefile`, (`make verbose` command), we can take a look at the linking process for the simple c program.

4 Takeaways:

Familiarize yourself with pointers and structures in C. Please try to play with compiling the given code in different C standards, using different GCC extensions and the memory section of different types of variables. There is extra code in the bonus directory for you to explore.