OBL1-OS

# The process abstraction

1. **Briefly describe what happens when a process is started from a program on disk. A mode switch from kernel- to user-mode must happen. Explain why this is necessary.**

The program that we want to run is located on a disk, therefore we need to be in kernel mode, since user mode does not have permission to read I/O devices. The kernel must load the program into its defined region of memory. Inside this memory region a heap and stack are created. Local variables are loaded into the stack when executing. The heap stores any dynamically allocated data the program needs.

After the program is loaded into memory from the I/O device (disk), we must switch from kernel- to user-mode. If we do not switch mode we would compromise the OS security. We do not want to be in kernel mode when a program is executing, since it would have complete control over the underlying hardware.

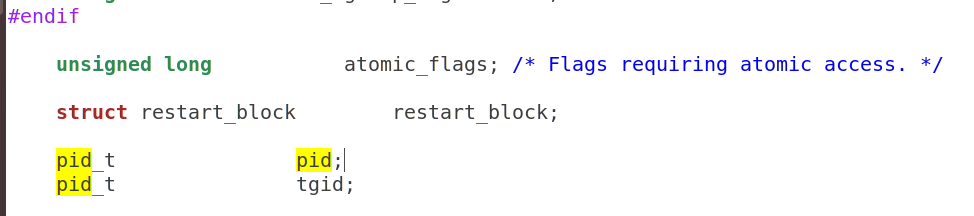
After this mode switch, the program is ready to run, but every instruction is checked by the kernel before it is loaded into a processor register. This is to prevent malicious instructions to run, so the kernel can raise an exception if necessary.

If the program wants to e.g. read from a disk to fetch some information on a file, it will need to make a system call to the kernel since it does not have the privileges to do such an action.

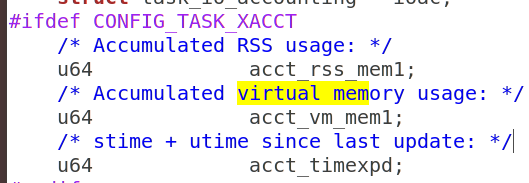
1. **Download the latest Linux kernel source code from https://kernel.org and unpack it. Use a web search engine to help identify the file in the source tree that contains the process descriptor structure (hint: its name is task struct). List the field name from this structure that:**

*This was found in the map include > linux > sched.h*

1. Stores the process ID

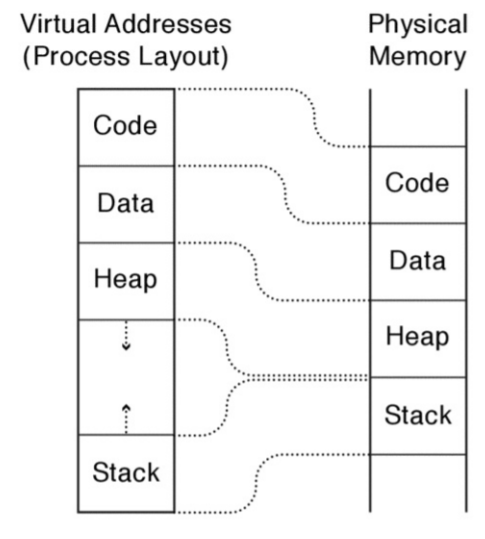


(b) Keeps track of accumulated virtual memory



# Process memory and segments

1. **Sketch the organisation of a process’ address space. Start with high addresses at the top, and the lowest address (0x0) at the bottom.**



1. **Briefly describe the purpose of each segment. Why is address 0x0 unavailable to the process?**

**Text / code segment**: Holds an executable image of the code, which is compiled.

**Data segment**: Holds global variables.

**Stack**: Holds local variables when they are needed when running the program.

**Heap**: Stores dynamically allocated data structure the program might need, like objects.

The address 0x0 is unavailable to the process because the null pointer is stored inside this address.

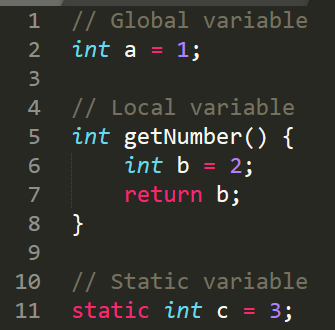
1. **What are the differences between a global, static, and local variable?**

When you declare a variable, you declare what data type the variable holds and where it should be stored. Where we declare this variable will define which scope the variable has, which will define where it will be stored.

**Global variables** are variables that is declared outside any function therefore they can be used “globally” anywhere in the code.

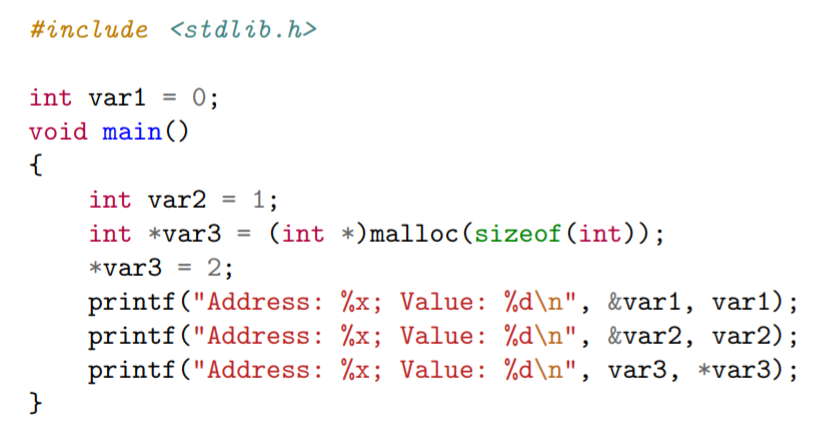
**Local variables** however, are declared inside a function, therefore it can only be used within that function.

**Static variables** can be declared anywhere in the code and be used anywhere in the code.



For example if I try to print variable b to the console outside the getNumber() function, I would get an error. This is because the variable is declared inside the function and is not available outside of it.

**Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.**



**Answer:**

**Var1**: Global variable, since it is declared outside of the main function. Belongs to the data segment.

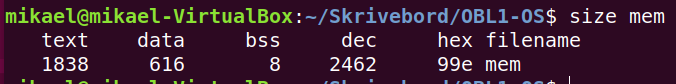
**Var2**: Local variable, since it is declared inside of the main function. Belongs to the stack.

**Var3**: Static variable, since it is a pointer. The size of the variable is an int. Belongs to the data segment.

# Program code

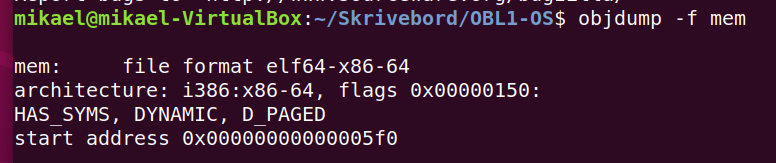
1. **Compile the example given above using gcc mem.c -o mem. Determine the sizes of the text, data, and bss segments using the command-line tool size.**

By using “size mem” in the terminal, we can find the sizes of the text, data, and bss segments.



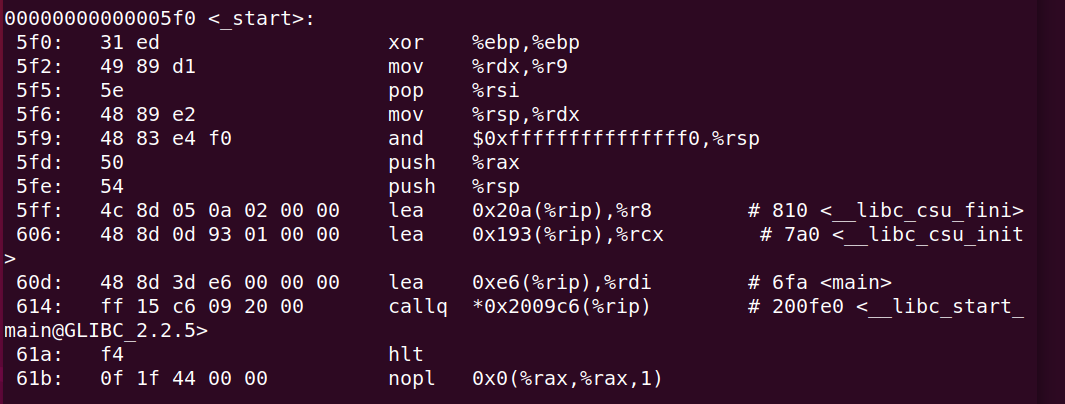
1. **Find the start address of the program using objdump -f mem.**

The start address of the program.



1. **Disassemble the compiled program using objdump -d mem. Capture the output and find the name of the function at the start address. Do a web search to find out what this function does, and why it is useful.**

The start function is useful because it will give the start address to the program on execution.



1. **Run the program several times (hint: running a program from the current directory is done using the syntax ./mem). The addresses change between consecutive runs. Why?**

The addresses change between consecutive runs because of randomization of addresses in memory. This is called “Address space layout randomization”. The main purpose of this is to combat viruses.

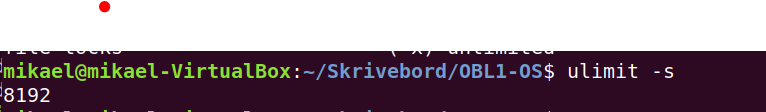
# The stack

1. **Compile the example given above using gcc stackoverflow.c -o stackoverflow.**



1. **Determine the default size of the stack for your Linux system. Hint: use the ulimit command (a web search or running the command ulimit --help will help find the appropriate command-line flags).**

To find the default size of the stack for my linux system I used the ulimit command with the -s flag. The size of my stack is 8192 kb (kilobytes).





1. **Run the program. Describe your observations and find the cause of the error.**

This program is an infinite loop. Function func() called in the main(), but in the func() function the func() is called, therefore it will keep calling itself infinitely. It will stop executing once the stack is full.

1. **Run the program and pipe the output to grep and wc -l: ./stackoverflow | grep func | wc -l What does this number tell you about the stack? How does this relate to the default stack size you found using the ulimit command?**

This command it will count the amount of lines the func() function outputs, which indicates how many times the func() function has been run.

Since number 523432 is the amount of lines the func() function writes, we need to divide it by 2 since we print 2 lines per function.

tells us roughly the amount of functions that can fit into the stack.



1. **How much stack memory (in bytes) does each recursive function call occupy?**

**(**8192 (kb) \* 1000) / 523432 / 2 = 31.3B ≈ 32 Bytes.

Each recursive function call occupies 32 bytes of stack memory.