1 The process abstraction

***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.***

When a process is started from a program on disk, The operating system sets aside a region of memory for the program. The operating system then copies the program instructions and data from the executable image into physical memory. The OS then sets the program counter to the first instruction of the program (memory address of the first instruction of the program is inserted in the program counter) and then the OS switches to user mode and starts executing instructions.

***A mode switch from kernel- to user-mode must happen. Explain why this is necessary.***

This is to prevent unrestricted access; in user mode a process cannot **directly** write/read outside its region in memory allocated by the operating system, if unchecked this could crash the whole system (if the process modifies the operating system’s own memory region) or other running programs by disrupting their memory. Reading outside your own memory region (from the perspective of a process) could also be malicious, this way a process can spy on other running processes. The example above was an example of memory resource management the operating system should deal with. The same applies for process management and other types of access to hardware, where mistakes can cost more than a simple OS crash. If the switch to user mode does not happen, a program will have unrestricted access to the hardware, and with great power comes great responsibility XD.

***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:***

1. ***Stores the process ID***

The field name can be found written as pid\_t pid. pid stands for process identification. The t at the end of the pid is to show that it’s a custom datatype defined with typedef.

***(b) Keeps track of accumulated virtual memory Use the Linux command-line tool top to explore other fields relating to running processes. Can you match them to field names in the process descriptor task struct? Name two such fields (besides those listed above).***

**u64 acct\_vm\_mem1**; accumulated virtual memory usage, u64 stands for unsigned 64 bits.

Table

Description automatically generated

**Int prio;** shown as PR next to user

**prev\_cputime, cputime\_expires**; Record the current running time (user mode and kernel mode), assuming its TIME+ in the figure.

Process memory and segments

The memory region allocated to a process contains the following segments.

 Text segment

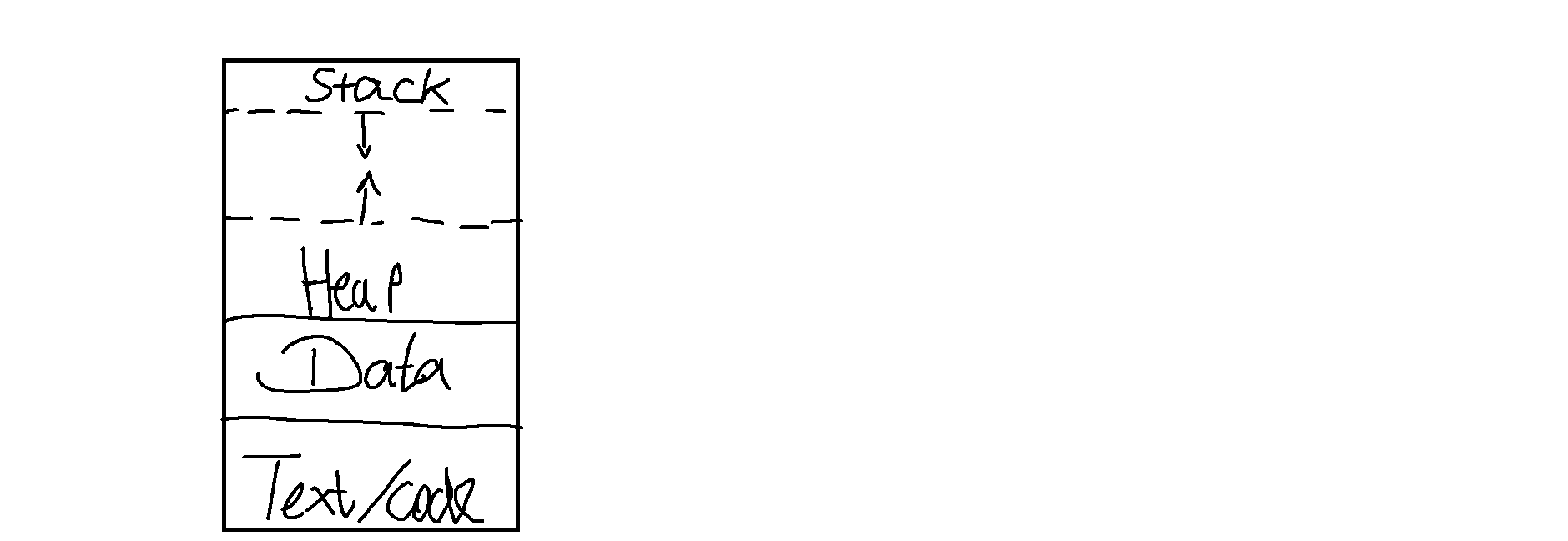
 Data segment

 Stack

 Heap

1. Sketch the organization of a process’ address space. Start with high addresses at the top, and the

lowest address (0x0) at the bottom.



**2. Briefly describe the purpose of each segment.**

Stack: where local variables are stored, grows downwards. Allocation and deallocation are automatic

Heap: Where dynamically allocated memory resides

Data segment: Contains global and static variables, has initialized and uninitialized sections

Text segment: contains read only executable instructions

**Why is address 0x0 unavailable to the process?**

Mostly for convention, depends on implementation as well. It is reserved for the null pointer.

“A null pointer has a reserved value that is called a null pointer constant for indicating that the pointer does not point to any valid object or function” - *https://www.ibm.com/docs/en/xl-c-aix/13.1.0?topic=pointers-null*

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

**Global** = can be accessed from anywhere within the program, global scope (across different .c files too). Example:

#include <stdio.h>

int countGlobal = 0;

int function1()

{

}

Here countGlobal is the global variable, stored in data segment.

**Static** = has a block scope within a program. static addresses could be global (with visibility to only current .c file), but they don’t need to be.

#include <stdio.h>

int function1()

{

static int count1 = 0; *// does not initialize the value of count1 to 0 at every function call*

count1++;

printf("count of function 1 is %d\n", count1);

}

Count1 is an example of a static local variable, it does not reset when you call the function again and can only be accessed within the function. Count1 is stored in data segment so that it does not reset and can be reused again whenever

**Local** = variables declared within a function or block

Example

#include <stdio.h>

int function1()

{

int count1 = 0; *// does not initialize the value of count1 to 0 at every function call*

count1++;

printf("count of function 1 is %d\n", count1);

}

Count 1 is an example of a local variable

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

**Text, letter

Description automatically generated**

Var 1 = global variable therefore stored in data segment.

Var2 = Local variable therefore stack.

Var3 = still a local variable, but since we use malloc (memory allocation), we are reserving space with the size of int on the heap. In other words, its dynamically allocated memory and therefore should be **found on the heap.**

**3 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.**

Text

Description automatically generated

Text = 1737

Data = 592

Bss = 8

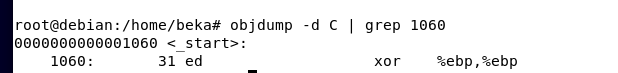
**2. Find the start address of the program using objdump -f mem.**

**Text

Description automatically generated**

Start address = 0x1060 in hexadecimal

**3. 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 initializes the program runtime to then invoke the main function in the program

**4. 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?**

**Text

Description automatically generated**

ASLR is the reason why, stack and heap addresses are not static, and local values are stored randomly on each start of the program.

The stack

1. **Compile the example given above using gcc stackoverflow.c -o stackoverflow.**
2. **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).**

**A screenshot of a computer

Description automatically generated with medium confidence**

8192 KB for the default stack size

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

**Text

Description automatically generated**

We see that frame address, which is the return address of the current function, and local address, which is the address of the local variable: both decrease by an offset of 0x20. The stack grows downwards.

Segmentation fault happens when you run out of stack memory, infinite or “too much” recursion can easily fill up the stack. We see that func() calls itself each time without any form of constraints to break the recursion and that is the root cause of the segmentation fault here.

**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 tells us how many hits for the text search “func” was found in the output of stackoverflow (the program). The more default stack size you have the more times the text func will pop up before it eventually crashes.

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

We found earlier that the stack grows downwards by 0x20 in hexadecimal, this converted to decimal is 32, and since each address identifies a single byte (8 bits), therefore 0x20 offset means 32 bytes of memory storage. Thus we can confirm that the func() function occupies 32 bytes of memory in the stack.