OBL1-OS

September 14, 2018

This is a mandatory assignment. Use resources from the course to answer the following questions. Take care to follow the numbering structure of the assignment in your submission. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine, for example running natively or virtually on your own PC, or by connecting to gremlin.stud.iie.ntnu.no over SSH (Secure Shell). Working in groups is permitted, but submissions must be individual.

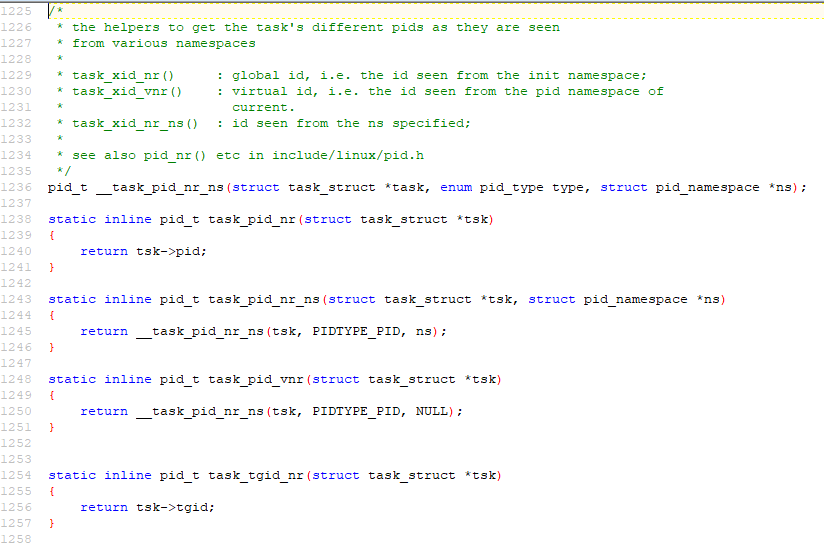
1 The process abstraction

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.

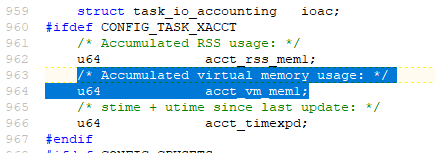
**A process is an instance of a program that is started. A mode switch from kernel- to user-mode is necessery to isolate the process from the rest of the computer.**

2. 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

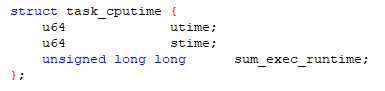


1. Keeps track of accumulated virtual memory



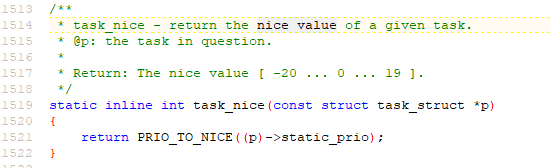
Use the Linux command-line tool top to explore other fields relating to running processes. Can you match them to field names in the process de- scriptor task struct? Name two such fields (besides those listed above).





**And**





2 Process memory and segments

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

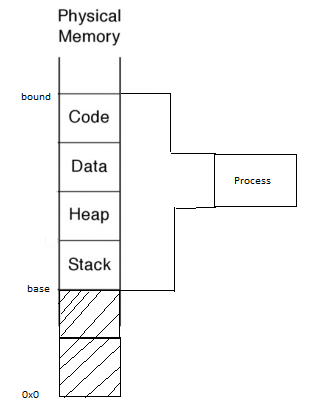
• Text segment

• Data segment

• Stack

• Heap

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



1. Briefly describe the purpose of each segment. Why is address 0x0 unavail- able to the process?

**The location 0x0 refers to a reserved null pointer, which means the pointer does not refer to a valid object. For example if it is out of bounds.**

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

* **Global variables are declared outside of functions and can be accessed from anywhere in the file.**
* **Local variables are declared inside a function, and the scope is within that same block.**
* **Static variables can be declared inside or outside a function, and their values do not change between function calls.**

1. Given the following code snippet, show which segment each of the variables

(var1, var2, var3) belong to.

#include <stdlib.h>

int var1 = 0;

void main()

{

int var2 = 1;

int \*var3 = (int \*)malloc(sizeof(int));

\*var3 = 2;

printf("Address: %x; Value: %d\n", &var1, var1); printf("Address: %x; Value: %d\n", &var2, var2); printf("Address: %x; Value: %d\n", var3, \*var3);

}

**Var1 is global because it is declared outside of the function.**

**Var2 is local because it is declared inside the function.**

**Var3 is a static variable because it uses the “malloc()”-function.**

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 = 546**

**Data = 16**

**bss = 8**

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

**0x00000000000005f0**

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.

**“xor” is a logical gate with two bits as input, and one as output. The table looks like this:**

A B | A XOR B

0 0 | 0

1 0 | 1

0 1 | 1

1 1 | 0

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?

**Because of a security feature called “Address Space Layout Randomization”.**

4 The stack

Consider the following C program:

#include <stdio.h>

#include <stdlib.h>

void func()

{

int localvar = 2;

printf("func() with localvar @ 0x%08x\n", &localvar);

printf("func() frame address @ 0x%08x\n", builtin\_frame\_address(0));

localvar++;

func();

}

int main()

{

printf("main() frame address @ 0x%08x\n", builtin\_frame\_address(0));

func();

exit(0);

}

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).

**8192 (KiB) is the default stack size.**

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

**Too many functions were called, resulting in a stackoverflow.**

4. 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?

**523444 refers to the number of newlines the program creates. There are two newlines in the loop as far as I can tell (one in each printf()).**

**52344 newlines / 2 = 261722 functions**

**(8192 \* 1024 B) / 261722 functions = 32 B per function**

1. How much stack memory (in bytes) does each recursive function call oc- cupy?

**32**