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.

When a process is started from a program on disk the operating system uses the process control block to control what the process has access to, who executed it and where its executable image resides on disk. Depending on what privileges the process has the dual mode operation switches modes. This is necessary to prevent anyone from accessing the hardware and keep malicious applications from writing directly to the disk, thus preventing the application from modifying the OS code and installing for example spyware.

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:

(a) Stores the process ID

- The field name of the process ID in the Linux kernel is : \_\_u32 ac\_ppid;

(b) Keeps track of accumulated virtual memory

- The field name of the accumulated virtual memory: \_\_u64 virtmem;

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).

* The field name of the user ID: \_u32uid;
* The field name of the resident memory usage: \_u64 coremem;

2 Process memory and segments

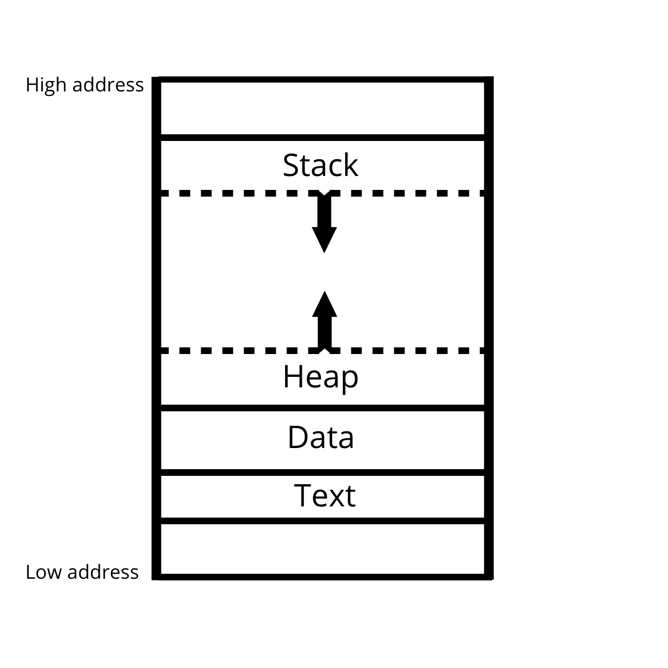
The memory region allocated to a process contains the following segments:

• Text segment

• Data segment

• Stack

• Heap

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

User level

Kernel level

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

* A text segment is one of the sections of a program in memory, which contains executable instructions.
* Data segment is a portion of an object file of a program that contains initialized static variables (global variables and static local variables).
* The stack is a memory region set aside by the operating system to hold the state of local variables during procedure calls.
* The heap is also a memory region set aside by the operating system for dynamically allocated data structures the program might need.
* The address 0x0 is unavailable to the process because it falls under the kernel space, which means the process doesn’t have the privileges in user mode to access it.

3. What are the differences between a global, static, and local variable? Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.

Global variables are declared outside any function (a group of statements that together perform a task), and they can be used on any function in the program. Meanwhile local variables are declared inside a function and can be only be used inside that function. A static variable has been allocated statically, this means its “lifetime” is the entire run of the program.

#include <stdlib.h>

int var1 = 0; //static variable

void main() // function

{

int var2 = 1; //local variable

int \*var3 = (int \*)malloc(sizeof(int)); //dynamic variable

\*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);

}

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 | data | bss |
| 1813 | 616 | 8 |

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

By using objdump -f mem I get that 0x0000000000000005f0 is the starting address

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 name of the function at the start address (0x0000000000000005f0) is <\_start>. <\_start> is the entry point of the program and it’s the address that the program jumps to when it starts.

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?

The addresses change because of the Address space layout randomization (ASLR), which is a computer security technique to prevent the exploitation of memory corruption vulnerabilities. The ASLR randomly arranges the address space positions for key data areas of a process.

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).

When I put in the command ulimit -s I get 8192 KB which is 8388608 Bytes. This is the same as 8 MB. This means that the default size of the stack in my Linux system is 8MB.

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

The program allocates different addresses to func(). We suddenly get the message “segmentation fault (core dumped)”, this means that the program is trying to access memory it doesn’t have access to. This means the program has reached the max size of the stack which was 8 MB.

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?

The “./stackoverflow | grep func | wc -l”-command tells us the amount of lines printed for func(). From the code, we can see that the program makes new variables until we get the message “segmentation fault (core dumped)”. Which means that when we get the “segmentation fault (core dumped)” message, the program is unable to make another variable. Segmentation fault (core dumped) means that the program doesn’t have any more memory it can access, because its restricted to the kernel. Since the program also prints a line to tell you where the new variable is located, we get twice as many lines as memory addresses. The program prints 523 324 lines before stopping, which means that the program can access 261 662 memory addresses before it reaches the stacks max size.

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

We have to divide 523324 by 2 to get the amount of recursive function calls that occur. This

523 324 / 2 = 261 662

Then we have to divide the size of the stack by the number of recursive function calls

8 388 608 / 261 662 = 32,06

this means that each recursive function call has the size of 32 bytes.