OBL1-OS

August 23, 2023

1. The process abstraction

1. When a process is started from a program on disk, the kernel must copy the program into the memory. The kernel must also set the program counter to the first instruction of the process and set the stack pointer to the base of the user stack. In other words, the kernel must initialize the PCB (Process Control block), a data structure that stores information about the process's state. After the kernel has allocated memory space for the for the process execution and initialized hardware resources, a mode switch from kernel- to user mode is necessary before execution. The reason this is necessary is because of security and isolation reasons. In the kernel-mode, the CPU has a privileged access to the hardware and can execute privileged executions. The kernel must always manage the system resources and provide resources where it is necessary. That is why it could be dangerous to run user processes in kernel-mode, since it could mess with the system and execute dangerous instructions. In user-mode, the CPU restrict access to hardware and is isolated from other user-processes. They can only access resources and perform operations that the kernel permits through system calls. In this way the mode switch ensures that user processes run with limited privileges.

2. The file that contains the process descriptor structure is located at: include\linux\sched.h

   a) Filed name of this structure that stores the process ID: pid
   b) Filed name of this structure that keeps track of accumulated memory: acct_vm_mem1

   Other field names in the process descriptor task_struct:

   - NI (Nice Value): Represents the nice value of the process.
   - RES (Resident Set Size): The portion of the virtual memory that is currently in physical RAM.

2. Process memory and segments

1. Here below is a sketch for the organization of a process' address space.

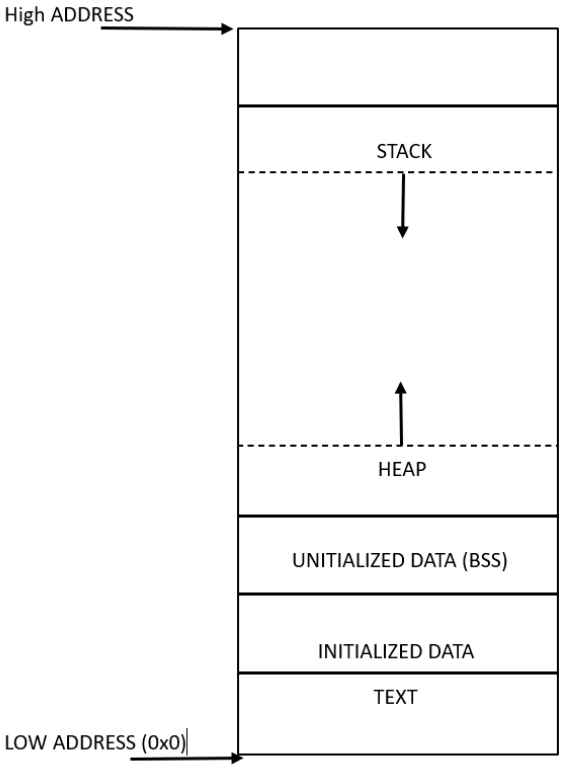

Figure 1: A sketch for the organization of a process' address space. Much taken from geeksforgeeks: https://www.geeksforgeeks.org/memory-layout-of-c-program/

2. *Stack*: The stack segment area is used for function call management and local variables. It stores information like function return addresses, local variables, and function call parameters. Each time a function is called, the address of where to return to and information about caller's environment and registers are saved on stack. With recursion a function calls itself, and a new stack frame is used so variables does not interfere with variables from another function instance. The stack grows downward, typically from high addresses to low addresses.

   Heap: The heap segment is used for dynamic memory allocation. It's where dynamically allocated data structures like arrays and objects are stored. It begins at the end of the BSS segment and grows upward to larger addresses from there.

   Uninitialized Data Segment: The uninitialized data segment is often just referred to as the bss segment (block started by symbol). This is the memory space for uninitialized variables that declared but haven't been assigned a value yet. Contains all global variables and static variables that are initialized to zero or do not have explicit initialization.

   Initialized Data Segment: Initialized data segment, also known as the Data Segment, is a portion of the virtual address space of a program. It contains the global and static variables that are initialized.

Text Segment: A text segment, also known as a code segment, is one of the sections of a program in an object file or in memory, which contains the executable instructions that the CPU executes. It is read-only, to prevent a program from modifying the instructions. It is typically mapped from an executable file and is shared among multiple instances of the same program.

Address 0x0: Address 0x0 is generally unavailable to the process because it's reserved for the operating system and hardware. Attempting to access memory at address 0x0 often leads to a segmentation fault or memory protection violation because this memory region is used by the operating system to manage process control structures and hardware interactions.

3. The differences between a global, static, and local variables.

   Global Variables: The global variables are declared outside of any function and are accessible throughout the entire program. They are created when the execution of a program begins and are lost when the program ends. Global variables also retain their values between function calls.

   Static Variables: Static variables can be declared both at the global scope and inside functions. When declared inside a function, they have limited visibility and are only accessible within the function. In similarity to a global variable, a static variable maintains its value between function calls, but their scope is confined to the declaring function.

   Local Variables: Local variables are declared within a specific function or block of code and are only accessible within that scope. They have a limited lifetime, as they are destroyed when the function exits. Local variables are not visible to other blocks of code unless they are passed as parameters.

By the code snippet illustrated below, we see three variables: var1, var2, and var3. Var1 belongs to the initialized data segment (DS), since it is a global variable, and since it is initialized. Var2 belongs to the stack segment because it is within a function. Var3 belongs to the heap because it is using the malloc() function. Memory allocated with malloc() and other heap-related functions usually resides in the heap segment. However, the pointer for var3 is stored on the stack because it is a locale variable, but the memory it points to resides in the heap.

```c
#include <stdio.h>
#include <stdlib.h>

int var1 = 0;
void main()
{
    int var2 = 1;
    int *var3 = (int *)malloc(sizeof(int)); // Note, since we are using malloc(), var3 will be a
                                            // pointer into the heap!
                                            // So the question is, where is the pointer stored?
    *var3 = 2;
    printf("Address: %x; Value: %d\n", &var1, var1);
    printf("Address: %x; Value: %d\n", &var2, var2);
    printf("Address: %x; Address: %x; Value: %d\n", &var3, var3, *var3);
}
```

3. Program code

1. The result from the commando line: "*size mem*":

```
jenscaa@jenscaa:~/os/2023/o1$ size mem
   text    data     bss     dec     hex filename
   1564     592       8    2164     874 mem
```

By reading the results we find that the text, data, and bss have the respectively sizes of 1564, 592, and 8 bytes.

2. The result from the commando line: "objdump -f mem":

```
jenscaa@jenscaa:~/os/2023/o1$ objdump -f mem

mem:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000001060
```

Here we find that the start address is 0x0000000000001060.

3. The result form the command line: "objdump -d mem":

```
0000000000001060 <_start>:
    1060:       31 ed                   xor    %ebp,%ebp
    1062:       49 89 d1                mov    %rdx,%r9
    1065:       5e                      pop    %rsi
    1066:       48 89 e2                mov    %rsp,%rdx
    1069:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    106d:       50                      push   %rax
    106e:       54                      push   %rsp
    106f:       45 31 c0                xor    %r8d,%r8d
    1072:       31 c9                   xor    %ecx,%ecx
    1074:       48 8d 3d ce 00 00 00    lea    0xce(%rip),%rdi        # 1149 <main>
    107b:       ff 15 3f 2f 00 00       call   *0x2f3f(%rip)        # 3fc0 <__libc_start_main@
    1081:       f4                      hlt
    1082:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    1089:       00 00 00
    108c:       0f 1f 40 00             nopl   0x0(%rax)
```

The name of the function at the start address is _start(). This function is responsible for setting up and prepare the initial environment for a program before calling the main function. This involves initializing the stack, call the _libc_start_main() function which initializes the C library. The function does also call the _init() function which preforms initialization before the main() function. Overall the _start() function is very useful since it plays a critical role in orchestrating a program's execution environment, interacting with the operating system, initializing runtime libraries, and ensuring program termination.

4. When running the program several times, the addresses are changing every time. This is because of the ASLR (address space layout randomization). The ASLR randomizes the memory layout that a program uses each time it runs. This is a safety measure from viruses and other malicious software that predicts the location of specific data structures or functions in memory.

4. The stack

1. Compiling done:

```
jenscaa@jenscaa:~/os/2023/o1$ sudo gcc stackoverflow.c -o stackoverflow
```

2. After running the “ulimit -s” command, I found that for my Linux system the default size of the stack is 8192 kilobytes.

```
jenscaa@jenscaa:~/os/2023/o1$ ulimit -s
8192
```

3. As a result of running the stackoverflow.c program I got a segmentation fraud, which is not surprising at all. The error is obvious, just as the program is named stackoverflow.c, we got a stack overflow. The reason is because of the recursive function func() that repeatedly calls itself without a termination or exit condition. This ultimately causes the stack overflow that leads to the segmentation fraud because it is trying to access memory that is not allocated for the stack. At last program is terminated.

4. By running the command: “./stackoverflow | grep func | wc -l”, I got a result of 523806.

```
jenscaa@jenscaa:~/os/2023/o1$ ./stackoverflow | grep func | wc -l
523806
```

This number indicates how many times a line with the string “func” is counted in the stackoverflow application. Since it is two lines that are written to stdout with “func” in each recursion, we simply have to divide the number with 2. As a result, the number of recursions is $523806/2 = 261903$. Compared to the default stack size I found by the ulimit -s command, which was 8192 kilobytes (8MB), we can tell that the number of recursions is limited by the stack segment limit.

5. $\frac{8192000\ bytes}{261903\ recursions} \approx 31\ bytes/recursion$. Each recursive function call occupies approximately 31 bytes in the stack memory.