Usman Ghafoorzai

OBL1-OS

1 The process abstraction

1.1

A process is defined as the execution of a program with restricted rights. To start the program running – the OS copies the program instructions and data from the program's executable image into the physical memory. The OS sets aside some memory for the execution stack to store the state of local variables during procedure calls. It also sets aside the heap (a region of memory) for any dynamically allocated data structures or objects the program might need. Beforehand, the OS itself must be loaded into memory, with its own stack and heap to copy the program into memory. Then, the OS ignores protection and starts running the program by setting the stack pointer and jumping to the first instruction of the program.

So, until the program is loaded into the physical memory, the OS is in kernel-mode. Once the program is loaded into the physical memory, the OS switches to user-mode and starts the process. The processor will fetch each instruction in turn, decode, and execute it.

The mode switch is necessary to prevent a process from doing any harm to other processes or to the OS itself and is called dual-mode operation. The OS kernel simulates, step by step, every instruction in every user process, instead of the processor directly executing instructions. Before each instruction is executed, the interpreter checks to see if the process had permission to do the operation in question. The instructor could allow legal operations while halting any application that overstepped its bounds. This is represented by a single bit in the processor status register to signify which mode the processor is currently executing in. In user-mode it checks to verify that the instruction is permitted before executing it. In kernel-mode the operating system executes with protection checks turned off.

Usman Ghafoorzai

1.2

Source code

```
GNU nano 7.2                                                printx.c *
#include <stdio.h>
#include <stdlib.h>

int main(int arg, char *argv[]) {
        // Checks wether 2 arguments is provided (argv[0] - program name, argv[1] - additional argument)
        if (arg != 2) {
                printf("Usage: %s <numer>\n", argv[0]);
                return 1;       // Exit with error code
        }

        // Convert arg from string to integer
        int X = atoi(argv[1]);

        // Check if the conversion was successful (atoi returns 0 if the input is invalid)
        if (X <= 0) {
                printf("Please  provide a valid positive integer for repetition count.\n");
                return 1;
        }

        // Prompt standard input
        char str[100];
        printf("Enter a string: ");
        fgets(str, sizeof(str), stdin);

        // Print the string X times
        for (int i = 0; i < X; i++) {
                printf("%s", str);
        }
        return 0;
}
```

Example 1

```
virtualmachine@virtualmachine:~/os-oving$ ./printx 3
Enter a string: Peanut
Peanut
Peanut
Peanut
virtualmachine@virtualmachine:~/os-oving$ 
```

Example 2

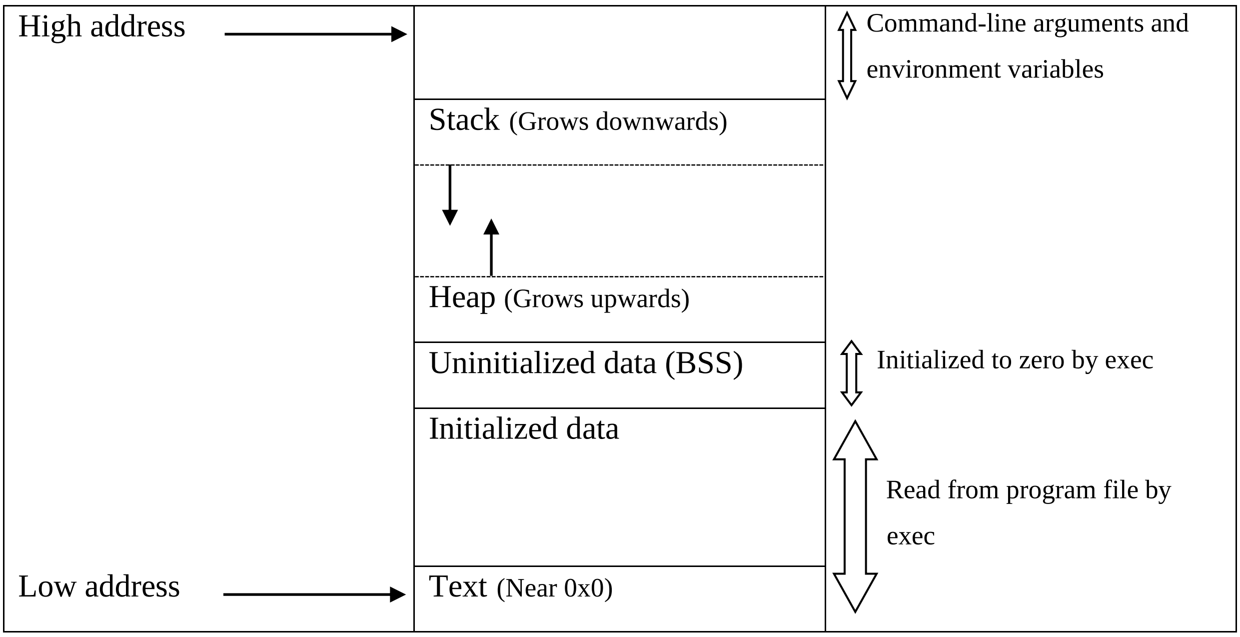
```
virtualmachine@virtualmachine:~/os-oving$ ./printx 3 1
Usage: ./printx <numer>
```

Example 3

```
virtualmachine@virtualmachine:~/os-oving$ ./printx a
Please  provide a valid positive integer for repetition count.
virtualmachine@virtualmachine:~/os-oving$ ./printx -1
Please  provide a valid positive integer for repetition count.
virtualmachine@virtualmachine:~/os-oving$ ./printx 0
Please  provide a valid positive integer for repetition count.
```

2 Process memory and segments

2.1



[Source]

2.2

1. The **text** segment/code segment contains the program's executable instructions and is usually read-only to prevent accidental modification – may be shared among processes to save memory. It is placed in memory to avoid overwriting by the stack or heap.

2. **Initialized data** segment stores global and static variables that have been initialized by the programmer. It has read-write permissions and includes:

Initialized Read-Only Area: Stores constant values, e.g., const char* s = “hello”.

Initialized Read-Write Area: Stores modifiable global/static variables, e.g., int debug = 1 or static int i = 10.

3. **Uninitialized Data Segment (BSS)** contains global and static variables initialized to zero or left uninitialized. This segment is set to zero by the compiler before execution. Examples include static int i; and int j;

4. The **stack** is a LIFO (last in, first out) structure used for function calls, local variables, and control flow. Grows downward (toward lower addresses) and is managed automatically by the program for storing return addresses, local variables, and function call information. Each function call creates a “stack frame” which allows for recursive function calls.

5. The **heap** segment is used for dynamic memory allocation and grows upward (toward higher addresses). It is managed by functions like malloc, realloc, and free, which may use system calls such as brk or mmap to adjust its size. The heap is shared by all libraries and dynamically loaded modules in a process.

The address **0x0** is marked as inaccessible (a null pointer) to help detect errors. As far as security is concerned, prevents a process from mistakenly or maliciously accessing sensitive memory. Furthermore, many programming languages use 0x0 as a null pointer value, indicating that a pointer does not reference any valid memory. Usually, if a program tries to access memory at 0x0, it is a sign of an error (such as a bug). The OS then terminates the process to prevent further problems [1].

2.3

Let us define the global, local, and static variables by its scope, lifetime and storage. Firstly, for **global variables**, it is accessible throughout the program, stored in the data segment, lifetime is the entire execution of the program. For **static variables**, have local or file scope but retain their value for the entire duration of the program, stored in the data segment. Lastly, for **local variables**, limited to the function or block scope, created and destroyed as needed, stored in the stack segment.

Based on the code snippet:

```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);
}
```

var1 belongs to global variable as it is defined outside of any function. Stored in the initialized data segment as it is explicitly initialized to 0. Meanwhile **var2** is a local variable since it is declared inside the main() function and is stored in the stack segment as it is a local variable. Lastly, **var3** – a pointer – is a local variable declared inside the main() function, therefore it is stored in stack segment. The memory to which it points is allocated using malloc(), so the actual memory block (the integer being pointed to and set to 2) is stored in the heap segment.

3 Program code

3.1

```
virtualmachine@virtualmachine: ~/os-oving
virtualmachine@virtualmachine:~/os-oving$ nano mem.c
virtualmachine@virtualmachine:~/os-oving$ gcc mem.c -o mem
virtualmachine@virtualmachine:~/os-oving$ size mem
   text    data     bss     dec     hex filename
   1512     592       8    2112     840 mem
virtualmachine@virtualmachine:~/os-oving$
```

3.2

```
virtualmachine@virtualmachine: ~/os-oving
virtualmachine@virtualmachine:~/os-oving$ objdump -f mem

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

virtualmachine@virtualmachine:~/os-oving$
```

3.3

```
Disassembly of section .text:

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@GLIBC_2.34>
    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 entry point of the program from a programmer's perspective is main, however _start is the usual entry point from the OS-perspective (that is, the first instruction that is executed after the program was started from the OS). It is useful since it is responsible for program initialization; sets up the environment required for the main function to execute, including initializing the stack, setting up the program's arguments, preparing the process's memory space and handling any required dynamic linking.

Thereafter it calls the main function of the program and handles the program exit with the returned status to properly terminate the program and return control to the OS.

Serves as a minimal entry point that is compatible with the system's executable format. Written in assembly language and is designed to be very small and efficient, this part is essential for the OS's loader, which needs to know where to start executing a program without the overhead of higher-level C runtime functions [2] [3].

3.4

```
virtualmachine@virtualmachine:~/os-oving$ ./mem
Address: 9fce3024; Value: 0
Address: 65eb702c; Value: 1
Address: 65eb7020; Value: -1613303136
virtualmachine@virtualmachine:~/os-oving$ ./mem
Address: 72b5024; Value: 0
Address: bfe3509c; Value: 1
Address: bfe35090; Value: 136008352
virtualmachine@virtualmachine:~/os-oving$ ./mem
Address: a7117024; Value: 0
Address: 54e6639c; Value: 1
Address: 54e66390; Value: -1477946720
virtualmachine@virtualmachine:~/os-oving$ ./mem
Address: d88b6024; Value: 0
Address: f89abb9c; Value: 1
Address: f89abb90; Value: -640572768
```

As shown in the screenshot above, the memory addresses printed by the program change between runs. This is due to **ASLR** – Address Space Layout Randomization – which is a security feature that randomizes the memory addresses used by a program every time it is run to prevent certain types of attacks, like buffer overflows, from exploiting predictable memory locations.

Usman Ghafoorzai

4 The stack

Consider the following C program:

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

void func()
{
    char b = 'b';
    /*long localvar = 2;
    printf("func() with localvar @ 0x%08x\n", &localvar);
    printf("func() frame address @ 0x%08x\n", __builtin_frame_address(0));
```

Page 2

```c
    localvar++;*/
    b = 'a';
    func();
}

int main()
{
    printf("main() frame address @ 0x%08x\n", __builtin_frame_address(0));
    func();
    exit(0);
}
```

4.2

```
virtualmachine@virtualmachine:~/os-oving$ ulimit -s
8192
```

4.3

```
virtualmachine@virtualmachine:~/os-oving$ ./stackoverflow
main() frame address @ 0xf356b880
Segmentation fault
```

The screenshot above displays that the program will terminate with a segmentation fault after running for a short period. This is due to an infinite recursion in the func() function, since it calls itself without any terminating condition and thus creating a new stack frame on the call stack. Each function call uses up stack space, and because the stack size is finite (as shown in ulimit -s command), the program will finally exhaust the available stack memory, resulting in a stack overflow. The program will then attempt to access memory outside of the stack’s allocated space. This results in a segmentation fault, since the system is preventing a program from corrupting memory outside its allowed boundaries.

4.4

Without printf()-statement in the func() function

```
virtualmachine@virtualmachine:~/os-oving$ ./stackoverflow | grep func | wc -l
0
```

With printf()-statement in the func() function

```
virtualmachine@virtualmachine:~/os-oving$ ./stackoverflow | grep func | wc -l
261578
```

4.5

Since the output form the command ./stackoverflow | grep func | wc -l is 261578, it means that the recursive function func() was called **261,578 times** before the program crashed due to a stack overflow. Moreover, we found that the total stack, according to the command ulimit -s, was **8192 KB**. To find out how much stack memory (in bytes) each recursive function call occupy, we simply compute **8192 * 1024 / 261587 = 32.07** bytes per call.

Sources

[1] https://www.geeksforgeeks.org/memory-layout-of-c-program/

[2] https://stackoverflow.com/questions/29694564/what-is-the-use-of-start-in-c

[3] https://adityakumawat502.medium.com/what-happens-before-the-main-function-executes-ea8f617bd62