1st Laboratory

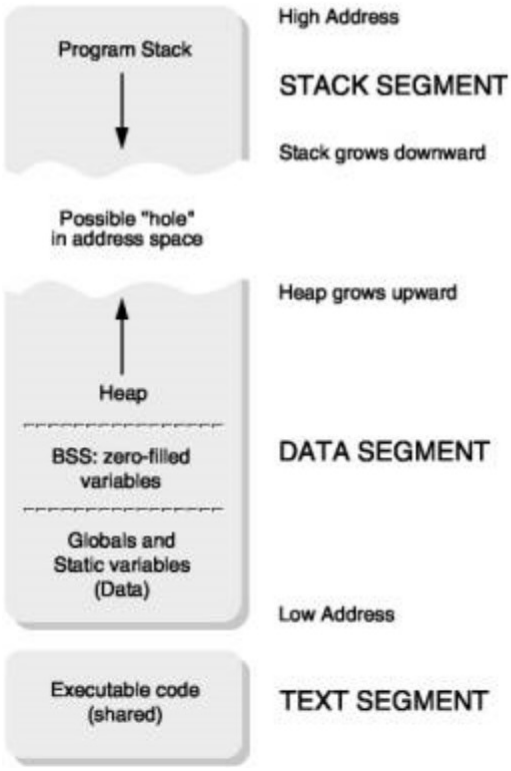
Memory management

Memory management subsystem is used by all other subsystems from the operating system: scheduling, I/O, file system, processes management, networking. Memory is a very important resource and this is the reason why we need very good algorithms for its management.

Memory management subsystem has to:

- Manage all physical memory zones (either free or used)
- Be able to offer memory for all the other subsystems when they need it
- Map virtual memory of different processes over physical pages
- .

Memory address space for a process



Every process has its own virtual address space. Even in the cases when 2 or more processes are sharing the same memory zone, the virtual address in each process is distinct, but it maps over the same physical memory.

In the image from the previous page you can see the virtual address space for a process. In modern operating systems, in the virtual address space of a process at the beginning the operating systems maps the kernel. In this laboratory we will talk only about the virtual address space from user mode used by a process.

The four important zones from the virtual space of a process are data zone, code zone, the stack and the heap. As you can see from the image presented above only the stack and the heap can grow. The heap and the stack are dynamic, while the code and data zone are static.

Text/code zone

The code segment (also knows as **`text segment`**) represents the instructions code of the application. The instruction pointer will reference addresses from the code zone. The CPU reads the next instruction, it decodes it and it executes it, after it increments the instruction pointer to read the next instruction. The code zone is usually a read only zone, to be sure that the application is not able to modify its own code if it has bugs or that any possible exploits do not alter the code. The code zone is shared between all the applications which are using the same program. This way, all processes sharing the same code are using the same physical memory for that code zone mapped at different addreses in the virtual psace,

Data zone

The data zone contains global variables and read only variables defined in the program. Depending on the data types there are a few different data zones subtypes.

.data

.data zone contains global variables and static variables declared in the program initialized to values different than zero For example:

```
static int a = 3;
char b = 'a';
```

.bss

`.bss` contains global and static variables declared in a program and uninitialized.
Before code execution this segment is initialized with 0. For example:

```
static int a;
char b;
```

.rodata

`.rodata` zone contains information which can only be read, but it can’t be modified.In this zone literas are stored. For example:

```
"Hello, World!"
```

In this segments also the global constants will be stored. Local variables declared with const will be placed on the stack, which is not a read only zone so they can be modified by a pointer which points to their address, although they are declared const. A special case is represented by local constant variables declared static which will be placed in .rodata also.

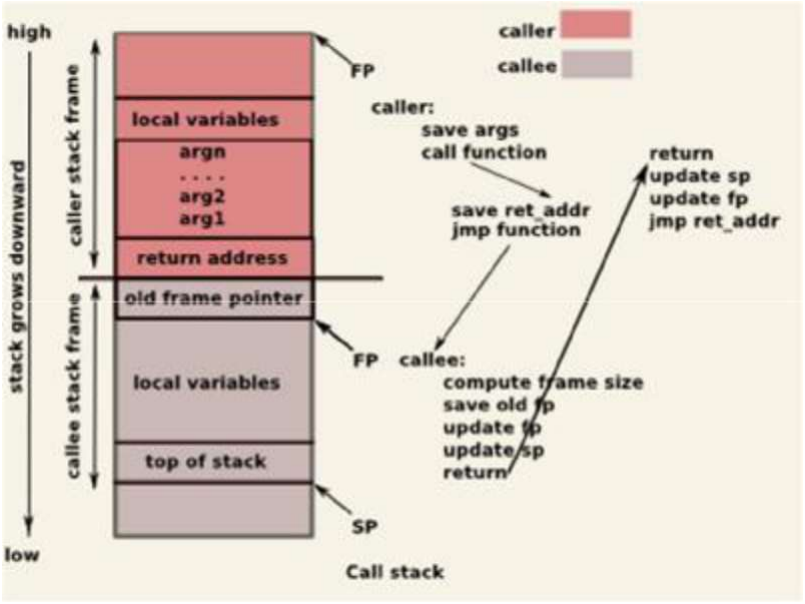
```
const int a;            /* în .rodata */
const char ptr[];       /* în .rodata */
void myfunc(void)
{
   int x;               /* pe stivă */
   const int y;         /* pe stivă */

   static const int z; /* în .rodata */
   static int p = 8;    /* în .data */
   static int q;        /* în .bss */

   ...
```

The stack

The stack is a dynamic region in a process, and is automatically handled by the compiler. The stack is used to store "stack frames". For each function call it will be created a stack frame.
A stack frame contains:

- Local variables
- Function arguments
- The return address.

Usually the stack extends downwards (from bigger virtual addresses to smaller virtual addresses) and the heap increases upwards. The stack increases on each function call and decreases after each function return.
The the below image you can observ the stack during a function call.



The heap

The heap is the memory zone dedicated to dynamic allocation. The heap is used to allocate different memory chunks which size is known only at the runtime.
Similar with the stack, the heap is a dynamic region which can expand its size. But in contrast with the stack, the heap is not handled by the compiler. The programmer has the duty to allocate the memory in heap, to keep track of it and to deallocate it after he/she is not using it. Frequent problems are represented by memory leaks – when a programmer allocates memory and forgot to deallocate it or when he/she accesses unallocated addresses.
In programming languages like Java, Lisp, etc, where there isn't "pointer freedom", the deallocation of the allocated space is done automatically by the garbage collector.

Practic.

1. Write a C program, which prints to console Hello World! Compile it! Run objdump -h executable_name and identify the zones described above. Find the stack zone and the heap zone. Identify the addresses from where each zone starts.
2. Create a loop in your program, to prevent it from ending. Run the command cat /proc/pid_number/maps – where pid_number is the pid of your process. Find where the stack and the heap are mapped. See that thare are other things mapped in your process, please identify what they are. Run now the command ldd executable_name – what do you observe and what is the relation with the mapping you previously observed?

Threads

We will remind you here some important functions for working with threads

pthread_create – creates a thread https://man7.org/linux/man-pages/man3/pthread_create.3.html

pthread_join – waits for a thread - https://man7.org/linux/man-pages/man3/pthread_join.3.html

mmap – used to allocate virtual memory https://man7.org/linux/man-pages/man2/mmap.2.html

sigaction – can be used to register a handler for a signal https://man7.org/linux/man-pages/man2/sigaction.2.html

With console command kill you can send a signal to a process.

Practic.

1. Write a C program which allocates a zone of virtual memory with only read only rights. Register a handler for SIGSEGV signal. Create a pointer to an address in the read only zone you just created and try to write at that address. Find in the handler registered to handle SIGSEGV signal the address where the access violation was produced and change the rights on the page with mprotect function.
2. Discussion – if I send a signal to a process which has more than one thread which thread will handle it? If a thread produces an invalid memory access, which thread will handle it?

Bibliography:

https://devarea.com/linux-handling-signals-in-a-multithreaded-application/#.Xksj6GgzZPY

Operating System Concepts 8th Edition – Abraham Silberschatz, Peter B. Gavin, Greg Gagne

https://ocw.cs.pub.ro/courses/so/laboratoare