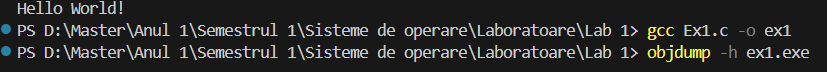
Laborator 1 – Rezolvare

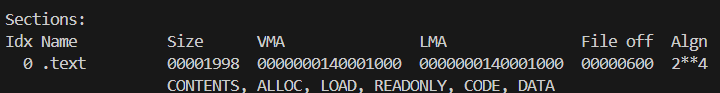
**Ex 1**. Write a C program, which prints to console *Hello World!*. Compile it! Run **objdump -h *executable\_name*** and identify the zones described above. Find the stack zone and the heap zone. Identify the addresses from where each zone starts.

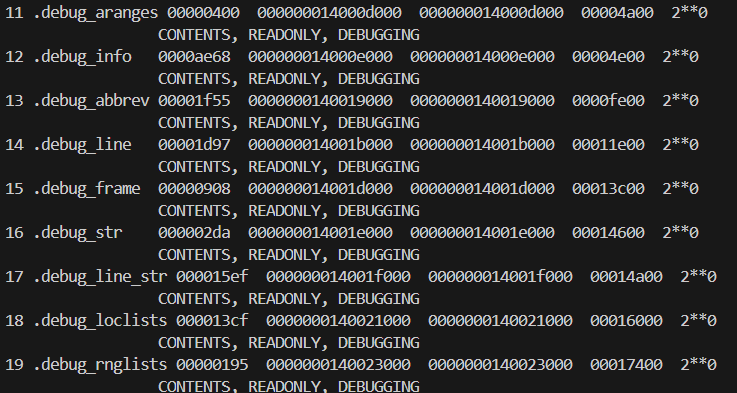
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Write a C program, which prints to console "Hello World!"  #include <stdio.h>  int main(int argc, char \*argv[]) {      printf("Hello World!\n");      return 0;  } |

Compilare:



Run **objdump -h *executable\_name***: objdump -h ex1 *→ Display the contents of the section headers using -h option (There can be various sections in an object file. Information related to them can be printed using -h option.)*





Ce reprezintă fiecare informație? → **Size**=dimensiunea secțiunii încărcate; **VMA**=**V**irtual **M**emory **A**ddress; **LMA**=**L**ogical **M**emory **A**ddress; **File off**=offset-ul secțiunii de la începutul fișierului; **Algn**=aliniere; **CONTENTS, ALLOC, LOAD, READONLY, DATA**=flags (dacă o secțiune este LOADED sau READONLY sau etc...). ([Sursă](https://www.thegeekstuff.com/2012/09/objdump-examples/))

Zona de Stack (.eh\_frame) și cea de Heap (.dynamic) nu pot fi identificate (sunt create dinamic în timpul rulării) deoarece programul nostru nu conține *stack frames* (care apar atunci când avem apeluri la funcții) și nici *alocări dinamice* (zona heap e dedicată pentru această problemă). (Sursă: Materialul de Laborator)

Zonele Identificate:

* Text/Code Zone: 0000000000001060
* .data: 0000000000004000
* .bss: 0000000000004010
* .rodata: 0000000000002000

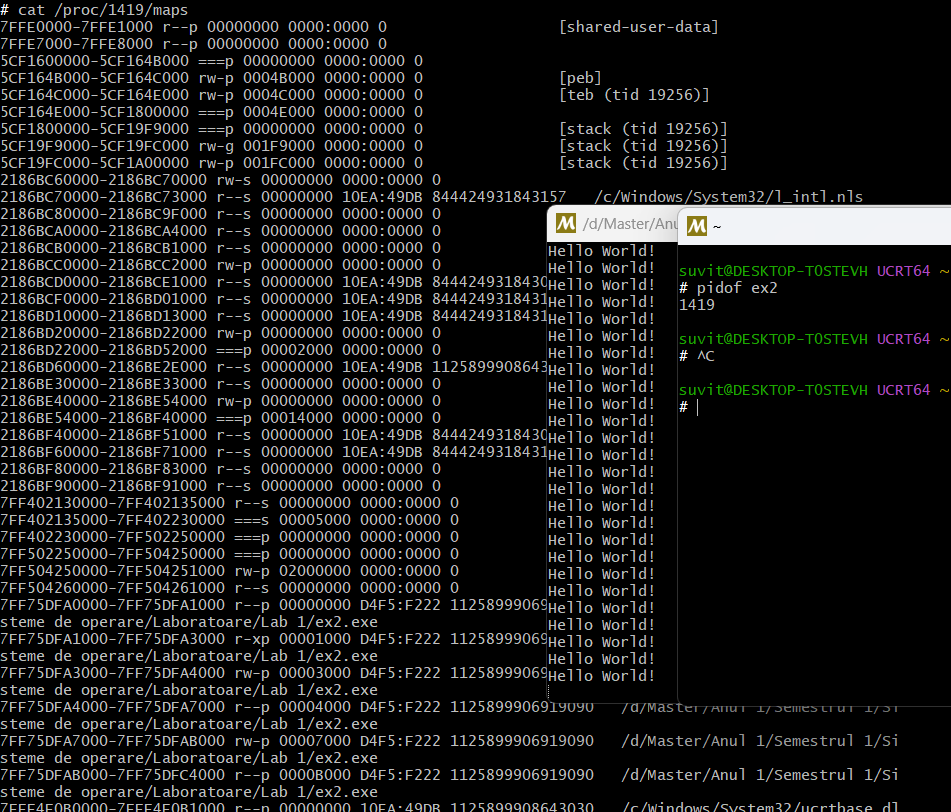
([Sursă](https://refspecs.linuxfoundation.org/LSB_1.2.0/gLSB/specialsections.html)).

**Ex 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?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Create a loop in your program, to prevent it from ending  #include <stdio.h>  int main(int argc, char \*argv[]) {      while(1) {          printf("Hello World!\n");      }      return 0;  } |

Pentru a afla PID-ul: **pidof** ***nume\_executabil*** ([Sursă](https://www.2daygeek.com/check-find-parent-process-id-pid-ppid-linux/))

În timp ce rulăm exercițiul 2, deschidem un nou terminal, aflăm PID-ul și rulăm comanda **cat /proc/*pid\_number*/maps**.

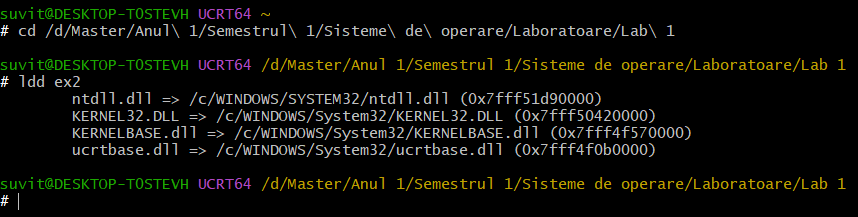


Semnificația pentru fiecare coloană:

* ***address***: adresa de început și cea de final a regiunii procesului
* ***perms (permissions)***: permisiuni de acces (r-read; w-write; x-execute; s-shared; p-private)
* ***offset***: Dacă regiunea a fost mapată dintr-un fișier (folosind *mmap*), acesta este offset-ul din fișierul de unde începe maparea. Dacă memoria nu a fost mapată dintr-un fișier, este doar 0.
* ***dev (device)***: Dacă regiunea a fost mapată dintr-un fișier, acesta este numărul de dispozitiv major și minor (în hex) în care se află fișierul.
* ***inode***: Dacă regiunea a fost mapată dintr-un fișier, acesta este numărul fișierului.
* ***pathname***: Dacă regiunea a fost mapată dintr-un fișier, acesta este numele fișierului (*libc* și *ld* sunt librării). Acest câmp este gol pentru regiunile mapate anonime. Există, de asemenea, regiuni speciale cu nume precum [heap], [stack] sau [vdso].
  + [heap]: mapată la adresa 7fffd7e37000-7fffd7e58000
  + [stack]: mapată la adresa 7fffdfd3b000-7fffe053b000
  + [vdso]: (virtual dynamic shared object → folosit de apelurile de sistem pentru a comuta la modul kernel) mapată la adresa 7fffe09b4000-7fffe09b5000

([Sursă](https://stackoverflow.com/questions/1401359/understanding-linux-proc-pid-maps-or-proc-self-maps))

Comandă: **ldd *nume\_executabil*** → *allows users to view an executable file’s shared object dependencies*.



Comanda ne arată librăriile de care depinde programul și adresa virtuală din memorie a acestor librării (unde sunt încărcate). ([Sursă1](https://man7.org/linux/man-pages/man1/ldd.1.html), [Sursă2](https://linuxhint.com/use-ldd-command-in-linux/))

Față de comanda anterioară, se poate remarca că adresa de memorie e diferită → fiecare librărie are mapată în memorie mai multe zone de memorie.

**Ex 3**. 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.

#include <stdio.h>

#include <sys/mman.h> //pt mmap, mprotect -> man mmap în terminal pt a acesa manual page

#include <signal.h> // pt sigaction -> SIGSEGV

#include <unistd.h> // pt \_exit

#include <stdlib.h> // pt coduri erori

// Atunci când primim semnalul, ar trebui să facem următoarele

void handler(int sig, siginfo\_t \*info, void \*ucontext) {

// Unde s-a produs accesarea ilegală

printf("Încercare de accesare ilegală a memorie la adresa: %p\n", info->si\_addr);

// Funcție care schimbă permisiunile pentru a putea să scriem

mprotect(info->si\_addr, 4096, PROT\_WRITE);

}

int main() {

// Alocare Memorie Virtuală

/\*

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

addr = The starting address for the new mapping. If addr is NULL, then the kernel

chooses the (page-aligned) address at which to create the mapping;

this is the most portable method of creating a new mapping.

length = length of the mapping (which must be greater than 0) --> 4096 = sizeof(int)

prot = the desired memory protection of the mapping --> PROT\_READ = Pages may be read.

flags = determines whether updates to the mapping are visible to other processes mapping

the same region, and whether updates are carried through to the underlying file.

--> MAP\_PRIVATE = Create a private copy-on-write mapping.

--> MAP\_ANONYMOUS = The mapping is not backed by any file; its contents are initialized to 0.

The offset argument should be zero.

fd = file descriptor --> 0

offset = starting the addr at defined offset --> 0

\*/

int \*ptr = mmap (NULL, 4096, PROT\_READ, MAP\_PRIVATE | MAP\_ANONYMOUS, 0, 0);

// Register a handler for a signal

/\*

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

------------------------------------------------------

If SA\_SIGINFO (This flag is meaningful only when establishing a signal handler.)

is specified in sa\_flags, then sa\_sigaction (instead of sa\_handler) specifies the

signal-handling function. This function receives three arguments, as described:

void handler(int sig, siginfo\_t \*info, void \*ucontext)

{

sig = The number of the signal that caused invocation of the handler.

info = A pointer to a siginfo\_t, which is a structure containing further information about the signal.

ucontext = The structure pointed to by this field contains signal context information that was

saved on the user-space stack by the kernel.

}

\*/

struct sigaction s;

s.sa\_flags = SA\_SIGINFO;

s.sa\_sigaction = handler;

// Setup signal handler

if(sigaction(SIGSEGV, &s, NULL) == -1) {

perror("Sigaction Error");

\_exit(EXIT\_FAILURE);

}

// Încercăm să scriem (READ\_ONLY)

ptr[0] = 5; // Se cheamă semnalul și mergem în funcția handler

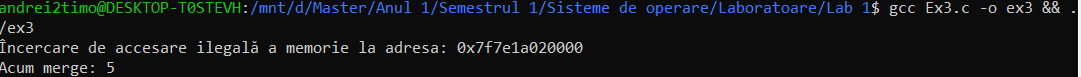
printf("Acum merge: %d\n", ptr[0]);

return 0;

}

([Sursă\_mmap](https://man7.org/linux/man-pages/man2/mmap.2.html), [Sursă\_sigaction](https://man7.org/linux/man-pages/man2/sigaction.2.html), [Sursă\_mprotect](https://man7.org/linux/man-pages/man2/mprotect.2.html))

Rezultat:



**Ex 4**. Discussion:

* *If I send a signal to a process which has more than one thread which thread will handle it?* → A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal. There is also the possibility of implementing a handling system in order for you, as the programmer, to choose a specific thread.
* *If a thread produces an invalid memory access, which thread will handle it?* → The thread itself using SIGSEGV.

([Sursă](https://unix.stackexchange.com/questions/225687/what-happens-to-a-multithreaded-linux-process-if-it-gets-a-signal), [Sursă2](https://stackoverflow.com/questions/7368113/signal-handling-in-c-with-process-and-2-threads-doesnt-work))