Executable and Linkable Format

ELF

- file format used in Linux for:
  - object files (gcc -c file.c)
  - **executable** (gcc -o file file.c)
  - shared libraries (gcc -shared -o file.so file.c)
  - core dumps
- program header table (PHT) describe *segments*, which contains one or more *sections*
- *loadable segments* provide the *execution* view of the object file

```
→ stack git:(master) ✗ file ./stack4
./stack4: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared
dID[sha1]=87beb67a2e7ad1edb7bf56520d4bbd2e07827c88, stripped
→ stack git:(master) ✗ xxd ./stack4 | head -n 1
0000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
```

- an executable begin a process when it is loaded in memory and executed

- Linux: *Executable and Linkable Format* (ELF): provides two interfaces: executable and linkable which are either described into the ELF header

- the *linkable* interface is not required for execution, therefore it will not be examined

- the *executable* interface is described by *program header* which are stored in a *program header table*

Tools

- objdump
- readelf
- file

print PHT using *readelf*

```
➜  stack git:(master) ✗ readelf -l ./stack4

Elf file type is EXEC (Executable file)
Entry point 0x8048380
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00660 0x00660 R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x00120 0x00124 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x000584 0x08048584 0x08048584 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

to be loaded in memory

PHT `aka` segment header table

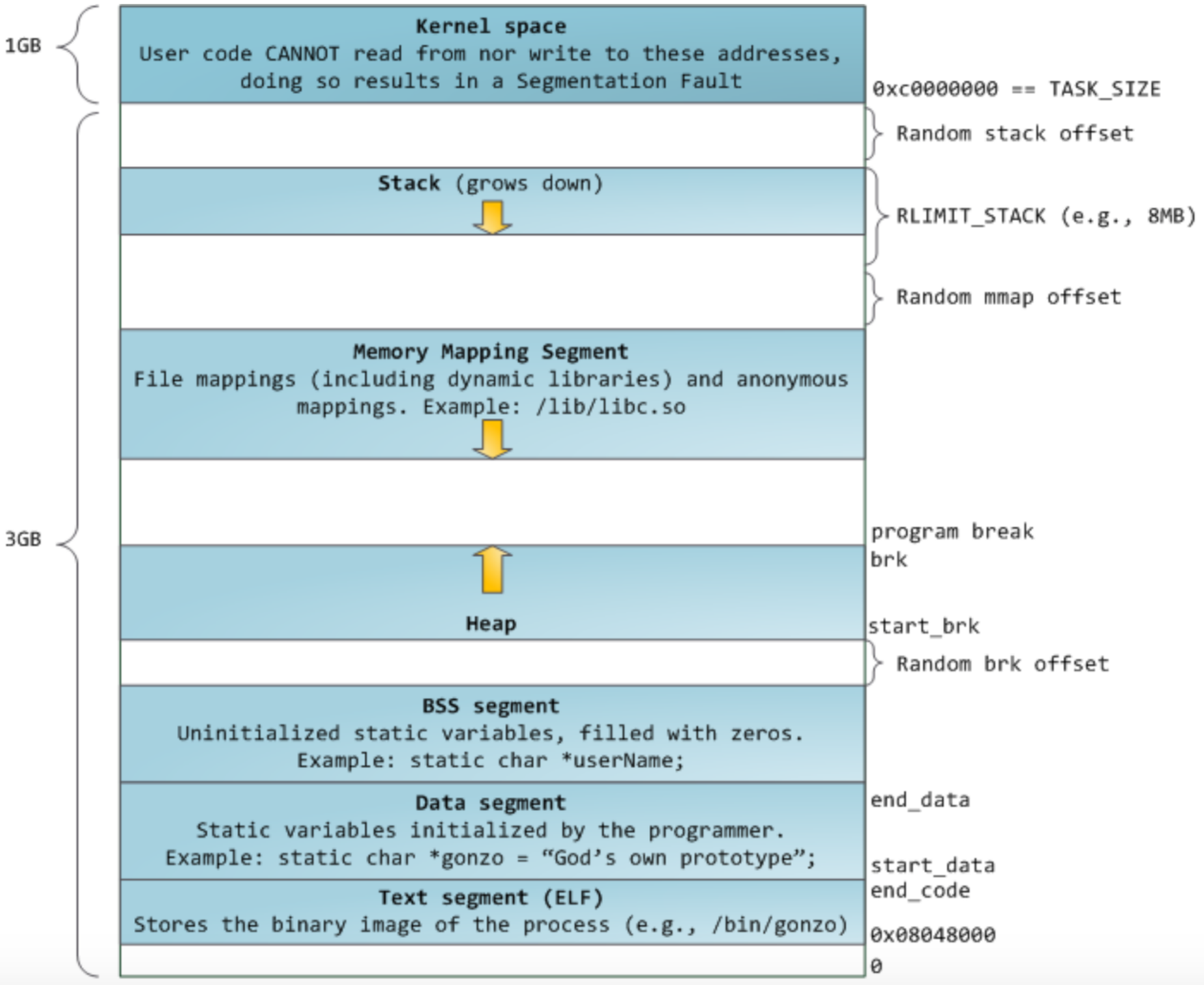
- to load a file in memory, program headers are used to provide informations like:
  - parts of file to be loaded into memory at runtime
  - locations of important data at runtime
- PHT describe *segments* as an offset from the start of the file and a size
- executable and shared object contains *segments*
- A *segments* is grouping of one or more sections

- neither SHT nor PHT have fixed address in memory

- to locate them we use the ELF header

```
→ stack git:(master) ✗ readelf -h ./stack4
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048380
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4460 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

- Process in memory are divided into three regions:
  - text: fixed by the program and includes code (instructions) and read-only data. loaded at fixed address, for i386 binary is 0x80480000 (except for Position Independent Executable)
  - data: contains initialized and uninitialized data (BSS) and static variables. it's size can be changed with the brk(2) system call
  - stack: we already know this data structure…


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
end_code
0x08048000
0

VMAs

- In Linux, a process' linear address space is organized in sets of virtual memory areas (VMAs)
- An object file's loadable segment corresponds to at least one VMAs
- The output from /proc/<pid>/maps lists the memory areas in this process's address space:

```
→ stack git:(master) ✗ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 524314                             /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 524314                             /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 524314                             /bin/cat
0113b000-0115c000 rw-p 00000000 00:00 0                                  [heap]
7fce195ac000-7fce19c8e000 r--p 00000000 08:01 660521                     /usr/lib/locale/locale-archive
7fce19c8e000-7fce19e49000 r-xp 00000000 08:01 399528                     /lib/x86_64-linux-gnu/libc-2.19.so
7fce19e49000-7fce1a048000 ---p 001bb000 08:01 399528                     /lib/x86_64-linux-gnu/libc-2.19.so
7fce1a048000-7fce1a04c000 r--p 001ba000 08:01 399528                     /lib/x86_64-linux-gnu/libc-2.19.so
7fce1a04c000-7fce1a04e000 rw-p 001be000 08:01 399528                     /lib/x86_64-linux-gnu/libc-2.19.so
7fce1a04e000-7fce1a053000 rw-p 00000000 00:00 0
7fce1a053000-7fce1a076000 r-xp 00000000 08:01 399504                     /lib/x86_64-linux-gnu/ld-2.19.so
7fce1a259000-7fce1a25c000 rw-p 00000000 00:00 0
7fce1a273000-7fce1a275000 rw-p 00000000 00:00 0
7fce1a275000-7fce1a276000 r--p 00022000 08:01 399504                     /lib/x86_64-linux-gnu/ld-2.19.so
7fce1a276000-7fce1a277000 rw-p 00023000 08:01 399504                     /lib/x86_64-linux-gnu/ld-2.19.so
7fce1a277000-7fce1a278000 rw-p 00000000 00:00 0
7ffdbf50a000-7ffdbf52b000 rw-p 00000000 00:00 0                          [stack]
7ffdbf530000-7ffdbf532000 r--p 00000000 00:00 0                          [vvar]
7ffdbf532000-7ffdbf534000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

PLT and GOT

- come vengono richiamate le funzioni contenute nelle *shared library*?

```
➜  stack git:(master) ✗ ldd ./got
        linux-gate.so.1 =>  (0xf76e8000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf751e000)
        /lib/ld-linux.so.2 (0xf76ea000)
```

- ldd mostra le *shared library* richieste dal binario *got*
- *Procedure Linkage Table (PLT)*
- *Global Offset Table (GOT)*

PLT and GOT

```
➜  stack git:(master) ✗ cat got.c
main()
{
  printf("GOT! \n");
}
```

prima dell'esecuzione

chiamata a puts

codice della funzione puts?

```
gdb-peda$ disass main
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     sub    esp,0x4
   0x08048423 <+6>:     mov    DWORD PTR [esp],0x80484d0
   0x0804842a <+13>:    call   0x80482f0 <puts@plt>
   0x0804842f <+18>:    leave
   0x08048430 <+19>:    ret
End of assembler dump.
gdb-peda$ disass puts
Dump of assembler code for function puts@plt:
   0x080482f0 <+0>:     jmp    DWORD PTR ds:0x804a00c
   0x080482f6 <+6>:     push   0x0
   0x080482fb <+11>:    jmp    0x80482e0
End of assembler dump.
gdb-peda$ printf "0x%x\n", *0x804a00c
0x80482f6
```

PLT and GOT

```
gdb-peda$ disass main
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     sub    esp,0x4
   0x08048423 <+6>:     mov    DWORD PTR [esp],0x80484d0
   0x0804842a <+13>:    call   0x80482f0 <puts@plt>
   0x0804842f <+18>:    leave
   0x08048430 <+19>:    ret
```

Impostiamo un breakpoint sulla prima istruzione del main ed eseguiamo

ora analizziamo il contenuto della GOT.
ora contiene l'indirizzo: 0xf7e76650

```
gdb-peda$ x/i 0x080482f0
   0x80482f0 <puts@plt>:        jmp    DWORD PTR ds:0x804a00c
gdb-peda$
   0x80482f6 <puts@plt+6>:      push   0x0
gdb-peda$
   0x80482fb <puts@plt+11>:     jmp    0x80482e0
gdb-peda$ printf "0x%x\n", *0x804a00c
0xf7e76650
gdb-peda$ disass 0xf7e76650
Dump of assembler code for function puts:
   0xf7e76650 <+0>:     push   ebp
   0xf7e76651 <+1>:     push   edi
   0xf7e76652 <+2>:     push   esi
   0xf7e76653 <+3>:     push   ebx
```

```
→ ~ cat /proc/31384/maps
08048000-08049000 r-xp 00000000 08:01 316714                             /home/r0x/lezioni/sicII/esercizi/stack/got
08049000-0804a000 r-xp 00000000 08:01 316714                             /home/r0x/lezioni/sicII/esercizi/stack/got
0804a000-0804b000 rwxp 00001000 08:01 316714                             /home/r0x/lezioni/sicII/esercizi/stack/got
f7581000-f7582000 rwxp 00000000 00:00 0
f7582000-f772a000 r-xp 00000000 08:01 556235                             /lib/i386-linux-gnu/libc-2.19.so
f772a000-f772c000 r-xp 001a8000 08:01 556235                             /lib/i386-linux-gnu/libc-2.19.so
f772c000-f772d000 rwxp 001aa000 08:01 556235                             /lib/i386-linux-gnu/libc-2.19.so
f772d000-f7730000 rwxp 00000000 00:00 0
f7746000-f7749000 rwxp 00000000 00:00 0
f7749000-f774b000 r--p 00000000 00:00 0                                  [vvar]
f774b000-f774d000 r-xp 00000000 00:00 0                                  [vdso]
f774d000-f776d000 r-xp 00000000 08:01 556237                             /lib/i386-linux-gnu/ld-2.19.so
f776d000-f776e000 r-xp 0001f000 08:01 556237                             /lib/i386-linux-gnu/ld-2.19.so
f776e000-f776f000 rwxp 00020000 08:01 556237                             /lib/i386-linux-gnu/ld-2.19.so
ffe3d000-ffe5e000 rwxp 00000000 00:00 0                                  [stack]
```

```
gdb-peda$ x/i 0x080482f0
   0x80482f0 <puts@plt>:        jmp    DWORD PTR ds:0x804a00c
gdb-peda$
   0x80482f6 <puts@plt+6>:      push   0x0
gdb-peda$
   0x80482fb <puts@plt+11>:     jmp    0x80482e0
gdb-peda$ printf "0x%x\n", *0x804a00c
0xf7e76650
gdb-peda$ disass 0xf7e76650
Dump of assembler code for function puts:
   0xf7e76650 <+0>:     push   ebp
   0xf7e76651 <+1>:     push   edi
   0xf7e76652 <+2>:     push   esi
   0xf7e76653 <+3>:     push   ebx
```