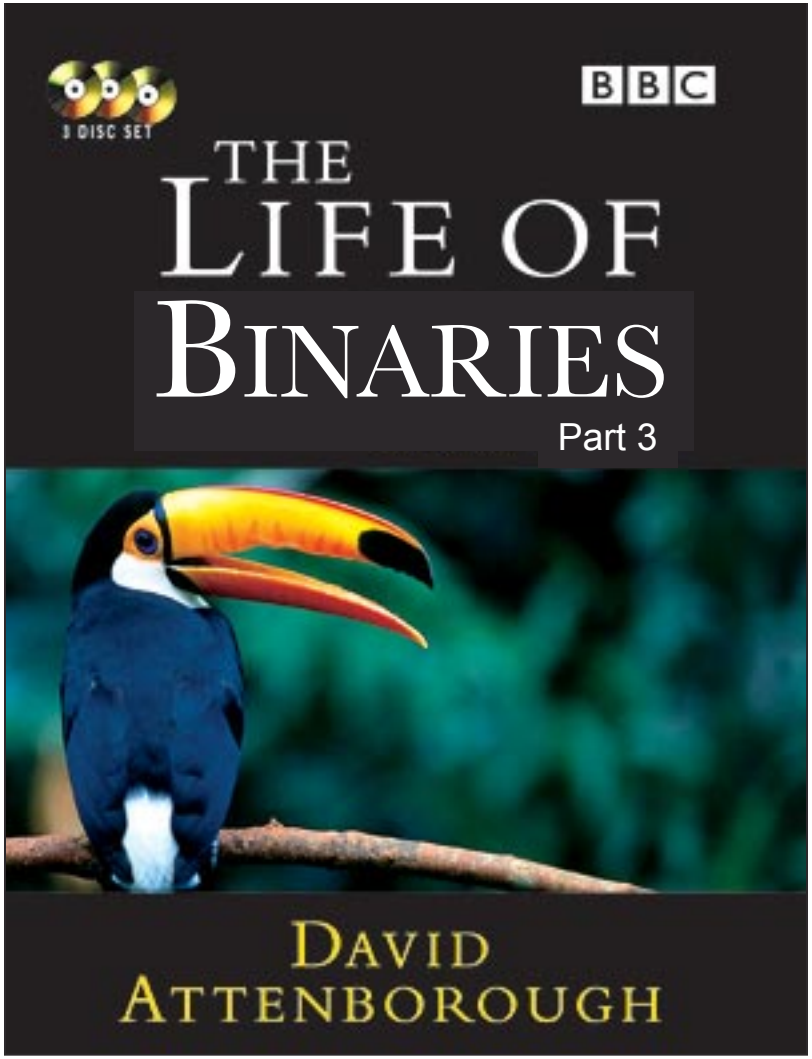



Xeno Kovah - 2010

xkovah at gmail

# History lesson

- “Sun Microsystems' SunOS introduced dynamic shared libraries to UNIX in the late 1980s. UNIX System V Release 4, which Sun co-developed, introduced the ELF object format and adapted the Sun scheme to ELF. ELF was clearly an improvement over the previous object formats, and by the late 1990s it had become the standard for UNIX and UNIX like systems including Linux and BSD derivatives.”

- from http://www.iecc.com/linker/linker10.html which has more fun info about linking

# Executable and Linkable Format (ELF)

- Official Application Binary Interface (ABI)
  http://**www.sco.com**/developers/devspecs/gabi41.pdf
- x86 supplement - http://**www.sco.com**/developers/devspecs/abi386-4.pdf
- Start at chapter 4


- April 24th 2001 draft update (supposedly widely used) http://refspecs.freestandards.org/elf/gabi4+/contents.html
- Oct 29th 2009 draft update http://www.sco.com/developers/gabi/2009-10-26/contents.html

# Your new new best friends: readelf, ldd, objdump,

- readelf will generally be included with any system which uses the ELF format.
- ldd can be used to display the shared libraries which an executable depends on (but then, so can readelf)
- We talked about objdump in the Intro x86 class as a good way to see the disassembly if you don't have IDA or you don't want to run GDB.

# Building Linux Executables/Libraries

- Normal dynamically linked executable:
  gcc -o <outfile> <source files>
- Normal statically linked executable
  gcc -static -o <outfile> <source files>
- Shared Object/Library (like a DLL) (more description here: http://www.ibm.com/developerworks/library/l-shobj/) "ld" is the linker
  gcc -fPIC -c -o lib<name>.o <name>.c
  ld -shared -soname lib<name>.so.1 -o lib<name>.so.1.0 -lc lib<name>.o
- Static Library (use "ar" to create a library archive file)
  gcc -c <source files>
  (the result of the -c will be a bunch of .o object files)
  ar cr lib<name>.a <object files ending in .o>
  (the "cr" are the "create" and "replace existing .o files" options)
  (subsequently when you want to link against your static archive, you can give the following options for gcc. The -l is lowercase l, the -L is optional if the file is already stored somewhere in the default path which will be searched by the linker)
  gcc -l<name> -L<path to lib<name>.a> -o <outfile> <source files>

# Field sizes

**Figure 4-2: 32-Bit Data Types**

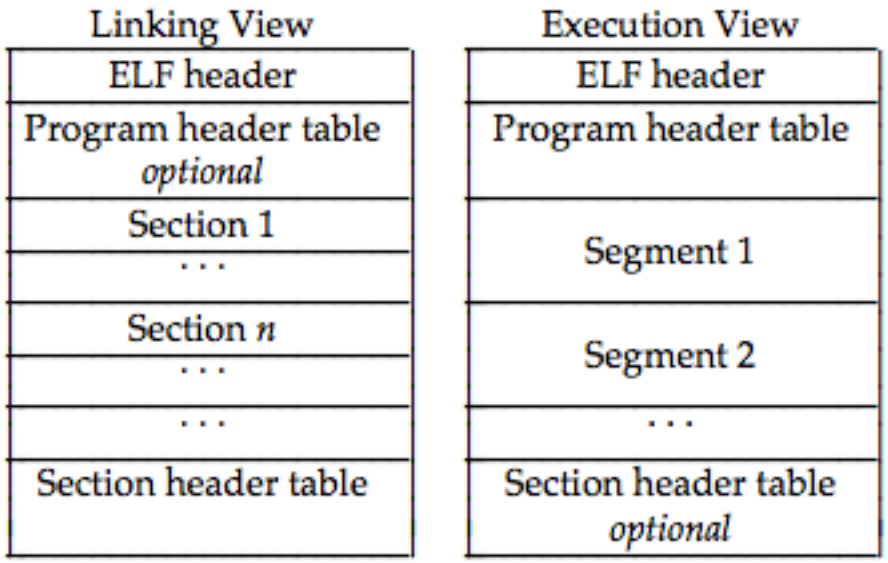
| Name | Size | Alignment | Purpose | aka |
|---|---|---|---|---|
| `Elf32_Addr` | 4 | 4 | Unsigned program address | unsigned int, DWORD |
| `Elf32_Half` | 2 | 2 | Unsigned medium integer | unsigned short, WORD |
| `Elf32_Off` | 4 | 4 | Unsigned file offset | unsigned int, DWORD |
| `Elf32_Sword` | 4 | 4 | Signed large integer | signed int, |
| `Elf32_Word` | 4 | 4 | Unsigned large integer | unsigned int |
| `unsigned char` | 1 | 1 | Unsigned small integer | |

Takeaway: everything except char and Elf32_Half are 4 bytes

But you also probably want to remember that _Addr is meant to be a virtual address, and _Off is meant to be a file offset.

# Overview

**Figure 4-1: Object File Format**




- Note: Linker cares about sections (like .text, .data etc), but multiple sections will get glommed together into an unnamed segment.
- PE combined sections too, but still the resultant section was called a section, and it still had a name.



All subsequent images are from the ABI

# ELF (File) Header

```
#define EI_NIDENT  16

typedef struct {
          unsigned char      e_ident[EI_NIDENT];
          Elf32_Half         e_type;
          Elf32_Half         e_machine;
          Elf32_Word         e_version;
          Elf32_Addr         e_entry;
          Elf32_Off          e_phoff;
          Elf32_Off          e_shoff;
          Elf32_Word         e_flags;
          Elf32_Half         e_ehsize;
          Elf32_Half         e_phentsize;
          Elf32_Half         e_phnum;
          Elf32_Half         e_shentsize;
          Elf32_Half         e_shnum;
          Elf32_Half         e_shstrndx;
} Elf32_Ehdr;
```



Just kidding :P

# ELF Header 2

from /usr/include/elf.h

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;            /* Section header string table index */
} Elf32_Ehdr;
```

# ELF Header 3

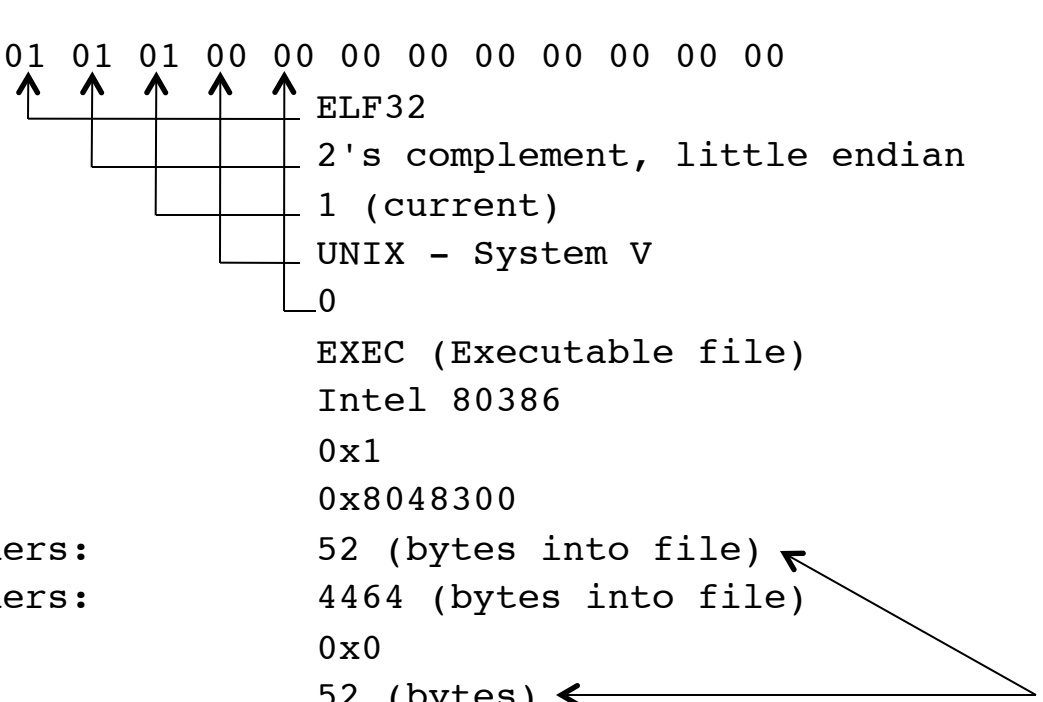
- **e_ident**[] starts with the magic number which is [0] = 0x7f, [1] = 'E', [2] = 'L', [3] = 'F'. This would be equivalent to the MZ signature at the start of PE files. Come on man, why's it always got to be about PEs? Hey, I'm just saying is all. Don't bite my head off. Whatever man, INFORMATION WANTS TO BE FREE! OPEN SOURCE FOREVER! Are you done? Yes. Good. There is other data encoded in e_ident, but we don't care about it that much and you can just look it up in the ABI if you're interested.
- **e_type** values that we care about are ET_REL (1) a relocatable file, ET_EXEC (2) an executable, ET_DYN (3) a shared object, and maybe ET_CORE (4)

# ELF Header 4

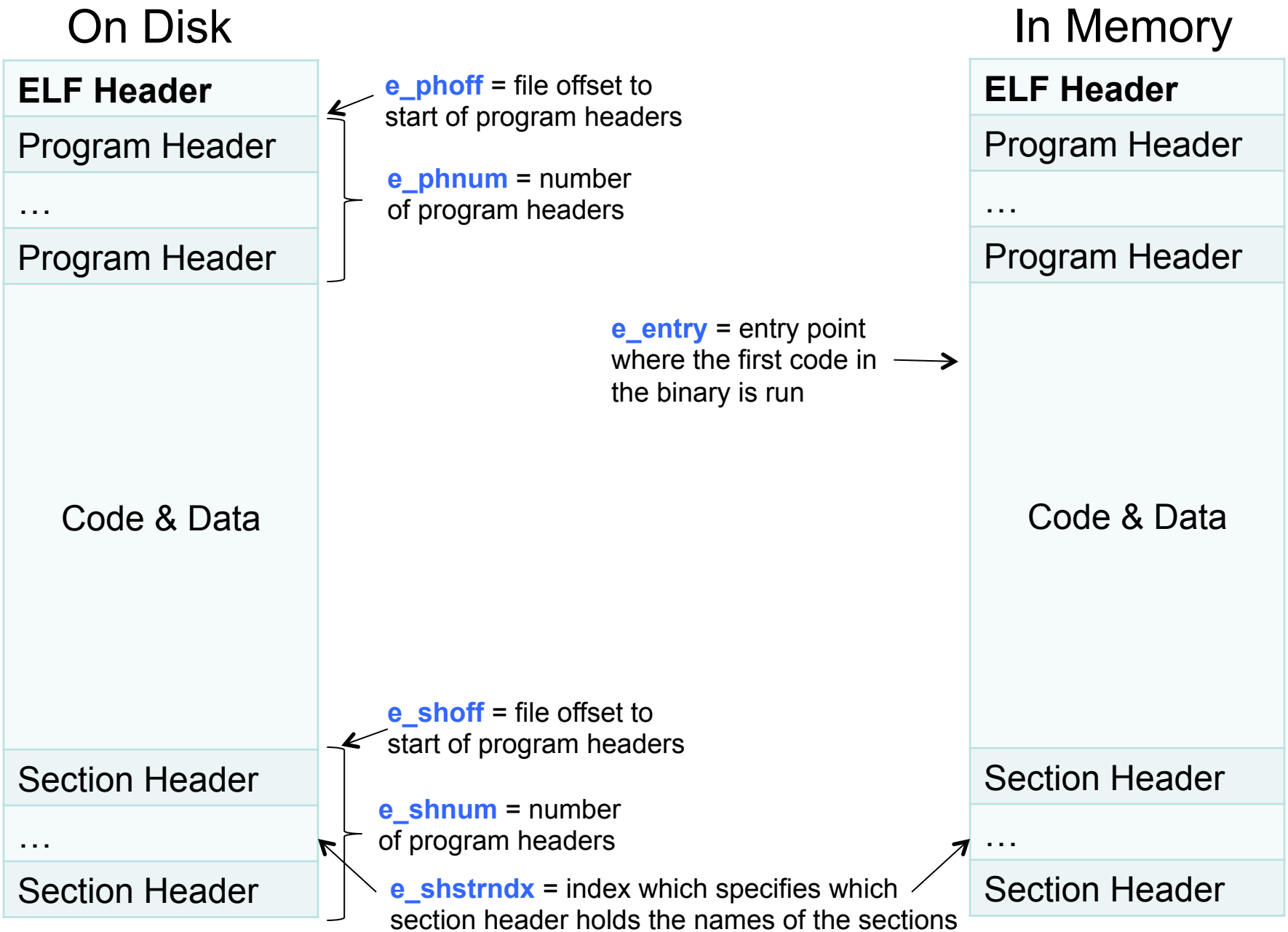
- **e_entry** is the VA (not RVA) of the entry point of the program. As before, don't expect this to point directly at main() be at the beginning of .text. It will typically point at the C runtime initialization code.
- **e_phoff** is a file offset to the start of the "program headers" which we will talk about later.
- **e_shoff** is a file offset to the start of the "section headers" which we will talk about later.
- **e_phnum** is the number of program headers arranged in a contiguous array starting at e_phoff.
- **e_shnum** is the number of program headers arranged in a contiguous array starting at e_shoff.
- e_ehsize, e_phentsize, and e_shentsize are the sizes of a single elf, program, and section header respectively. Unless something like a packer is messing with the format, for a 32 bit executable these should be fixed to 52, 32, and 40 bytes respectively.
- **e_shstrndx** was described somewhat confusingly in the spec but this is the index of a specific section header in the section header table which is the string table (which holds the names of the sections)

# Display ELF header: readelf -h

```
user@ubuntu:~/code/hello$ readelf -h hello
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:               0x8048300
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4464 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 26
```



# ELF Header Fields



# Program (segment) Header

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

# Program (segment) Header

```
typedef struct
{
  Elf32_Word    p_type;      /* Segment type */
  Elf32_Off     p_offset;    /* Segment file offset */
  Elf32_Addr    p_vaddr;     /* Segment virtual address */
  Elf32_Addr    p_paddr;     /* Segment physical address */
  Elf32_Word    p_filesz;    /* Segment size in file */
  Elf32_Word    p_memsz;     /* Segment size in memory */
  Elf32_Word    p_flags;     /* Segment flags */
  Elf32_Word    p_align;     /* Segment alignment */
} Elf32_Phdr;
```

# Program Header 2

- **p_type** has the following subset of values that we care about:
  - **PT_LOAD** is the most important. This specifies a chunk of data from the file which will be mapped into memory.
  - **PT_DYNAMIC** specifies a file/memory region which holds dynamic linking info.
  - **PT_INTERP** points a string which the loader uses to actually first load an "interpreter". The interpreter is then responsible for doing whatever with the program which asked for it to be invoked. However, in practice, for executables, the "interpreter" is the dynamic linker. And what it does is set everything up for dynamic linking.
  - **PT_PHDR** is just for if the binary wants to be able to identify its program headers
  - PT_TLS is Thread Local Storage again
  - For the rest RTFM
- An executable can run with only two PT_LOAD segments (as we shall see later with UPX), everything else is optional. (I feel like they can run with only one as well, but I haven't tried…that's good homework for you :))

# Program Header 3

- If you want to read about other things like PT_NOTE, see the manual, if you want to read about things like PT_GNU_STACK (http://guru.multimedia.cx/pt_gnu_stack/) or PT_GNU_RELRO(http://www.airs.com/blog/archives/189) there you go.
- NOTE TO SELF: I should probably go into that PT_GNU_STACK stuff since it's about executable stack.

# Program Header 4

- **p_offset** is where the data you want to map into memory starts in the file. But again, remember, only for PT_LOAD segments does data actually get read from the file and mapped into memory.
- **p_vaddr** is the virtual address where this segment will be mapped at.
- p_paddr is supposed to be the physical address, but "Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects." This is frequently set to the same thing as p_vaddr, but you can basically just ignore it.

# Program Header 5

- **p_filesz** is how much data is read from the file and mapped into memory. Or alternatively, how much data
- **p_memsz** is the size of the virtual memory allocation for this segment. If p_memsz is > p_filesz, the extra space is used for the .bss area. (Remember that conceptually the .bss section is all about allocating some space in memory for uninitialized variables which you don't need/ want to store in the file.)
- Also recall that the only way that the size in memory (IMAGE_SECTION_HEADER.Misc.VirtualSize) could be smaller than the size on disk (IMAGE_SECTION_HEADER.RawData) for PE files was due to the RawData being small and being aligned up to a multiple of the file alignment size. ELF has no notion of file alignment, therefore the following always is true: p_filesz <= p_memsz.

# Program Header 6

- **p_flags** are the memory permission flags. PF_X = 0x1, PF_W = 0x2, PF_R = 0x4, or any number of processor-specific things as long as the MSB is set (which can be checked by ANDing with PF_MASKPROC). So the lower 3 bits are RWX like normal UNIX filesystem permissions.
- **p_align** is the segment alignment. The segment must start on a virtual address which is a multiple of this value. For normal PT_LOAD segments, the alignment is 0x1000. In contrast, PE sections don't have to be 0x1000 aligned, so it's sort of like ELF enforces memory alignment while PE doesn't, and PE enforces file alignment while ELF doesn't.

# Display program header: readelf -l

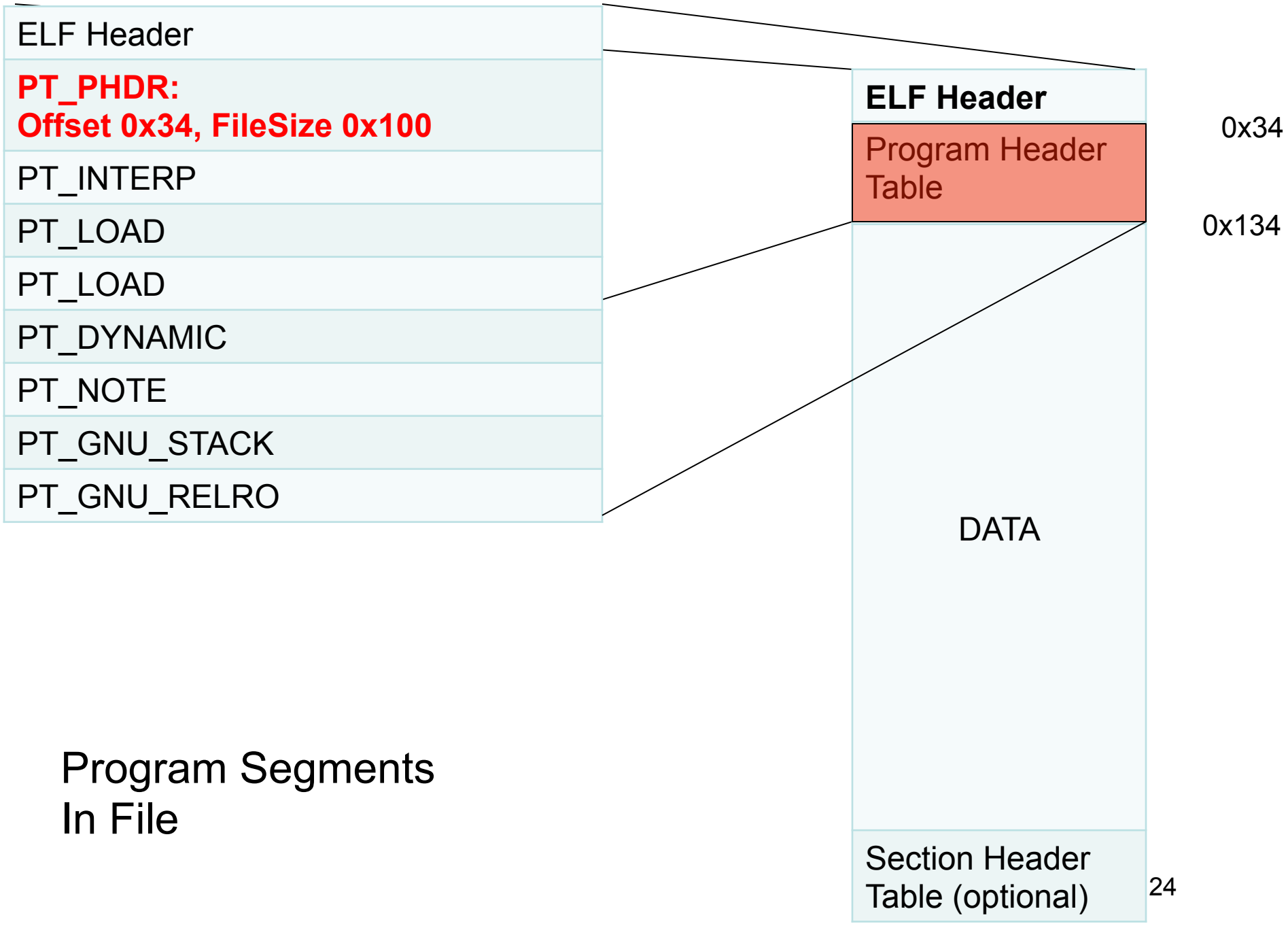
```
user@ubuntu:~/code/hello$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x8048300
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004a4 0x004a4 R E 0x1000
  LOAD           0x000f14 0x08049f14 0x08049f14 0x00100 0x00108 RW  0x1000
  DYNAMIC        0x000f28 0x08049f28 0x08049f28 0x000c8 0x000c8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f14 0x08049f14 0x08049f14 0x000ec 0x000ec R   0x1

<SNIP>
```

# readelf -l part 2

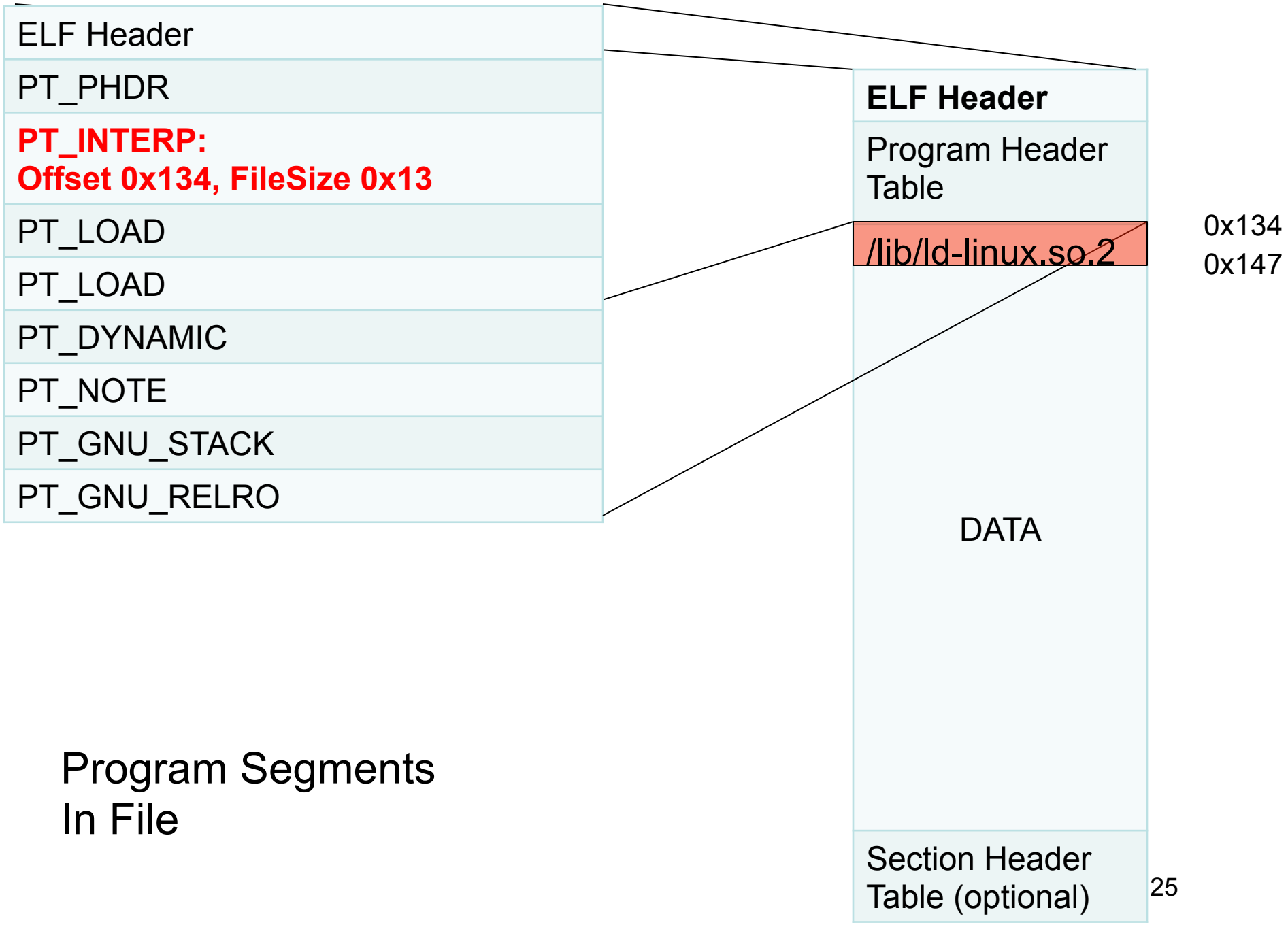
```
user@ubuntu:~/code/hello$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x8048300
There are 8 program headers, starting at offset 52
<SNIP>
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.version_r .rel.
  dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06
   07     .ctors .dtors .jcr .dynamic .got
```

# program headers from statically linked binary

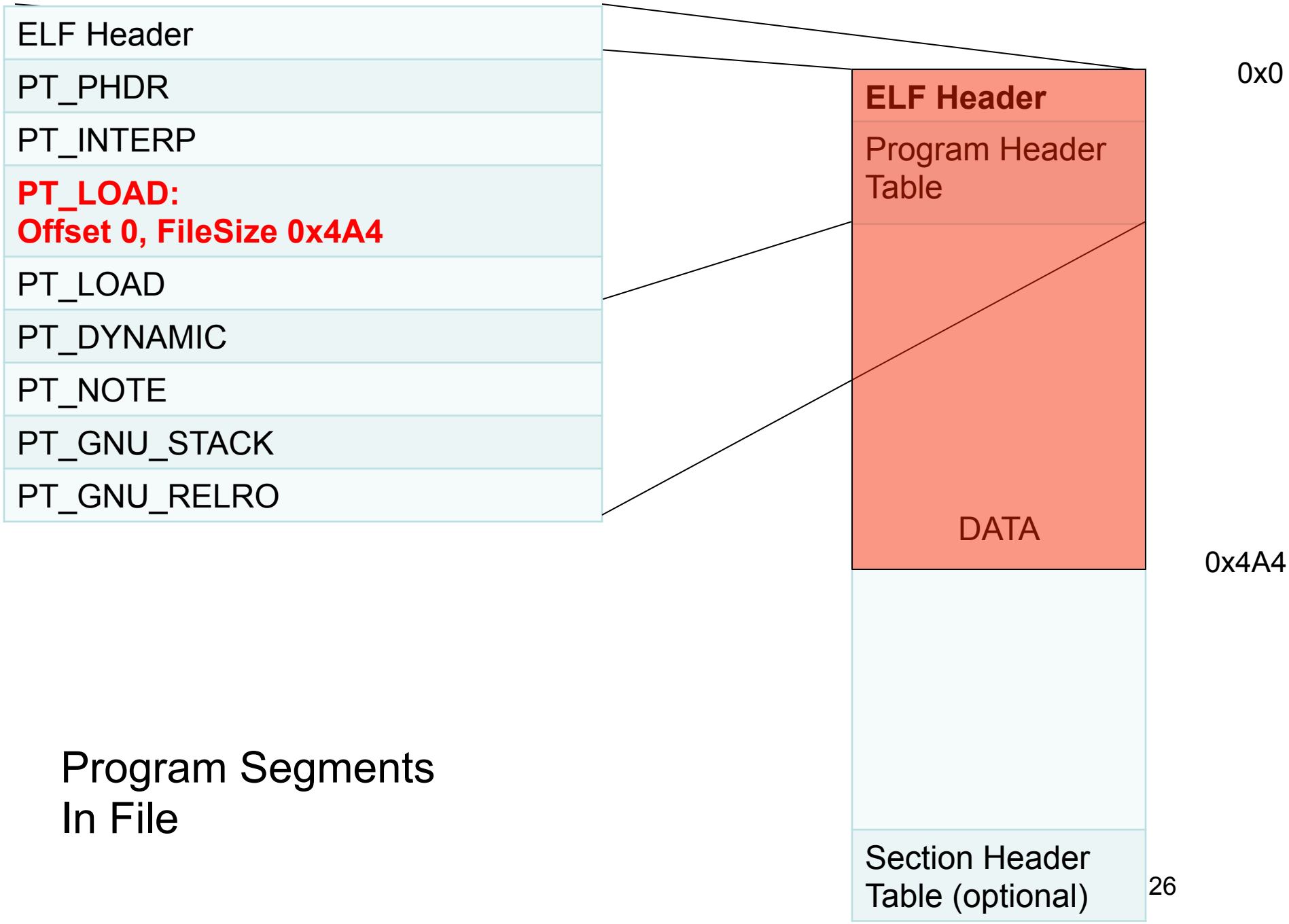
```
user@ubuntu:~/code/hello$ gcc -static -o hello-static hello.c
user@ubuntu:~/code/hello$ readelf -l hello-static

Elf file type is EXEC (Executable file)
Entry point 0x80481e0
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x851df 0x851df R E 0x1000
  LOAD           0x085f8c 0x080cef8c 0x080cef8c 0x007d4 0x02388 RW  0x1000
  NOTE           0x0000f4 0x080480f4 0x080480f4 0x00044 0x00044 R   0x4
  TLS            0x085f8c 0x080cef8c 0x080cef8c 0x00010 0x00028 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x085f8c 0x080cef8c 0x080cef8c 0x00074 0x00074 R   0x1

<SNIP>
```



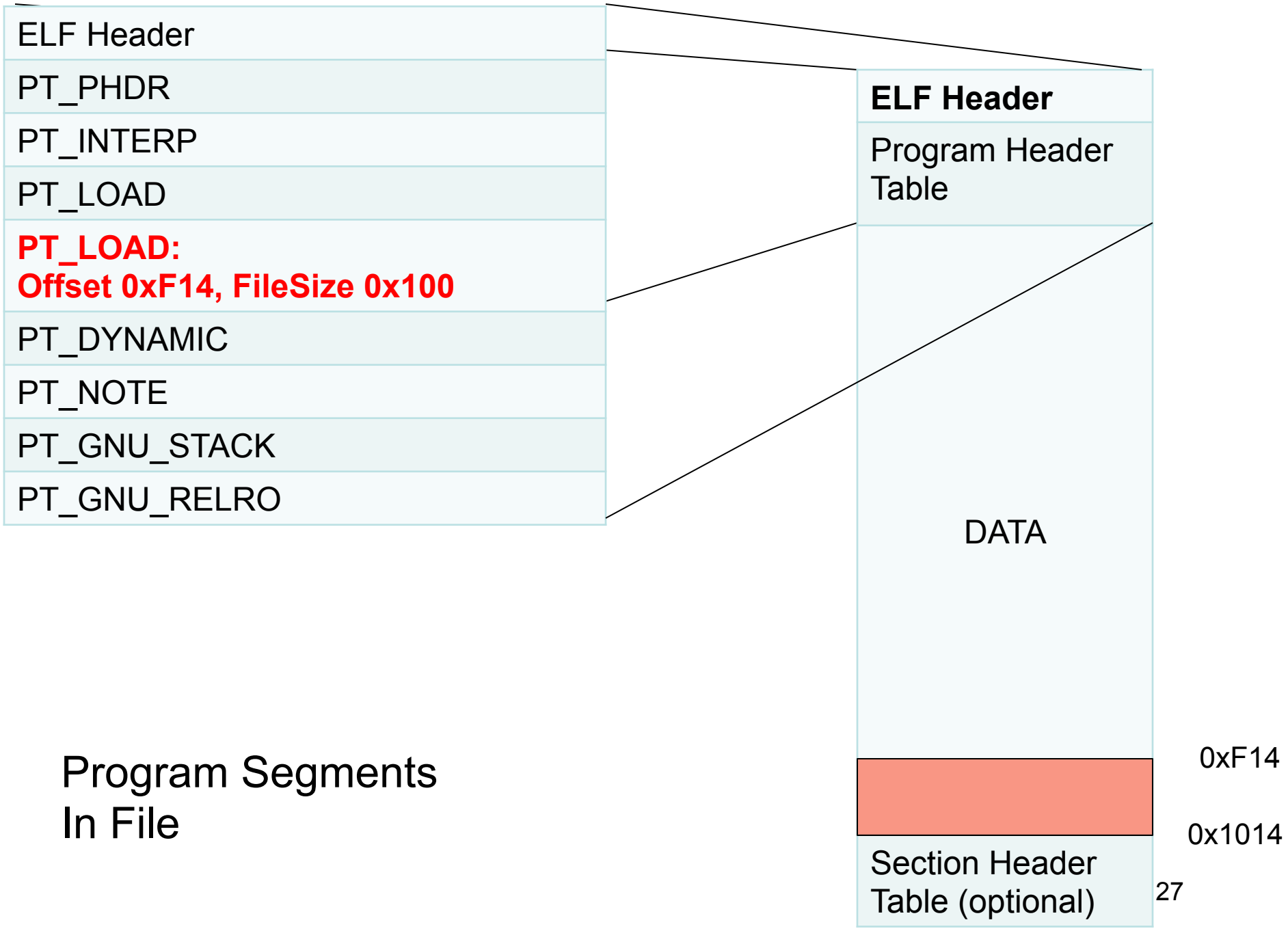

Program Segments
In File



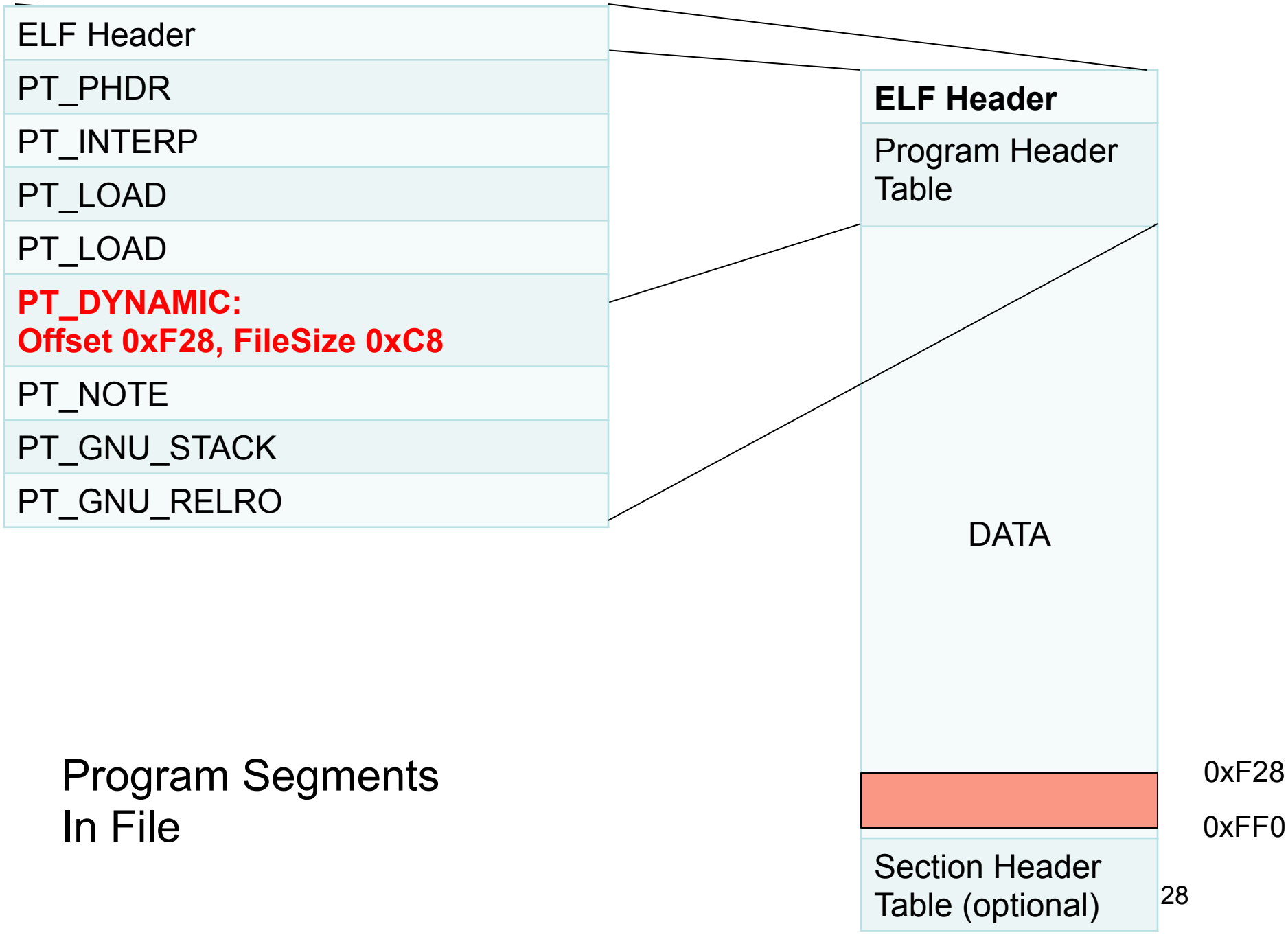

Program Segments
In File



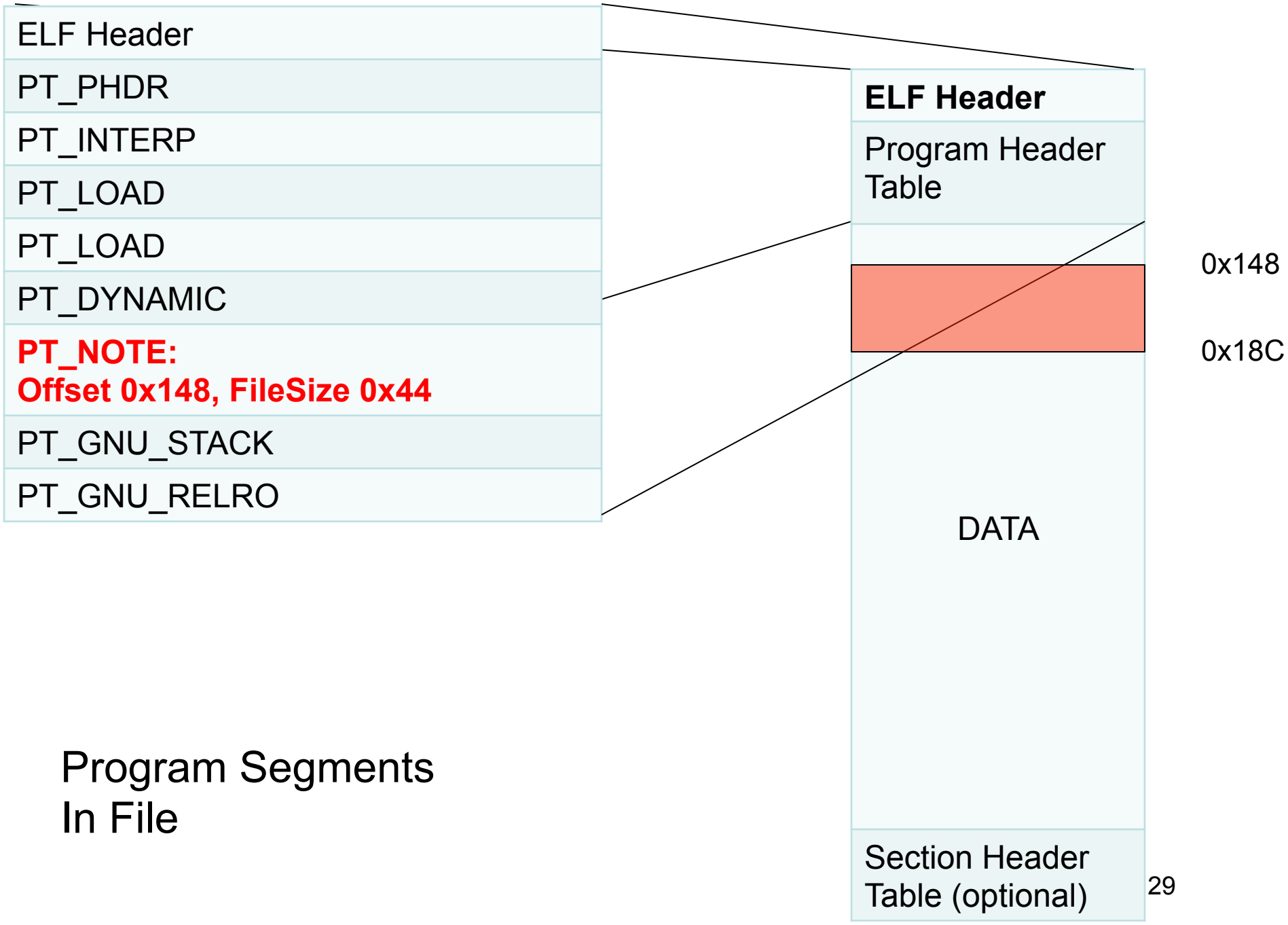

Program Segments
In File



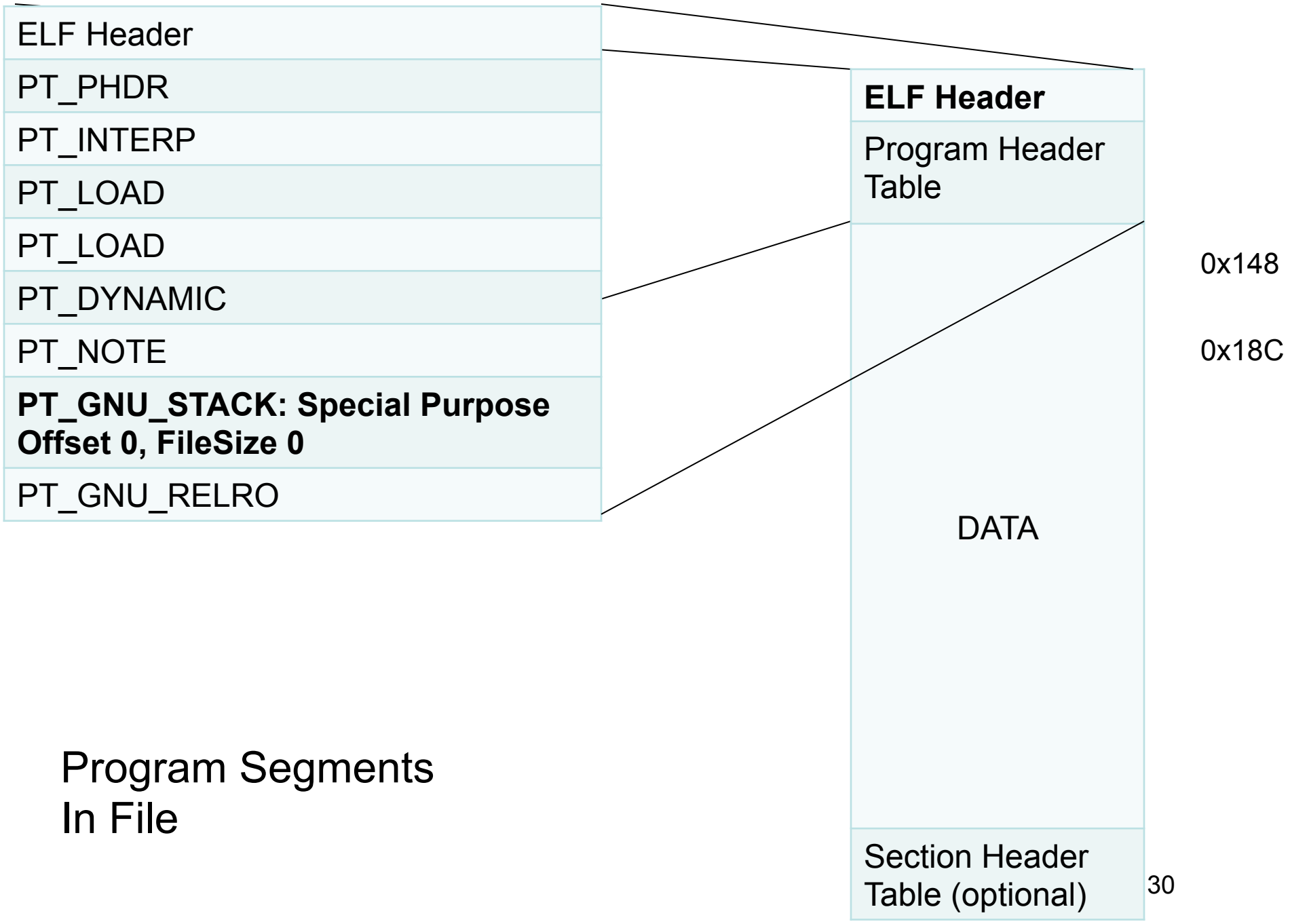

Program Segments
In File



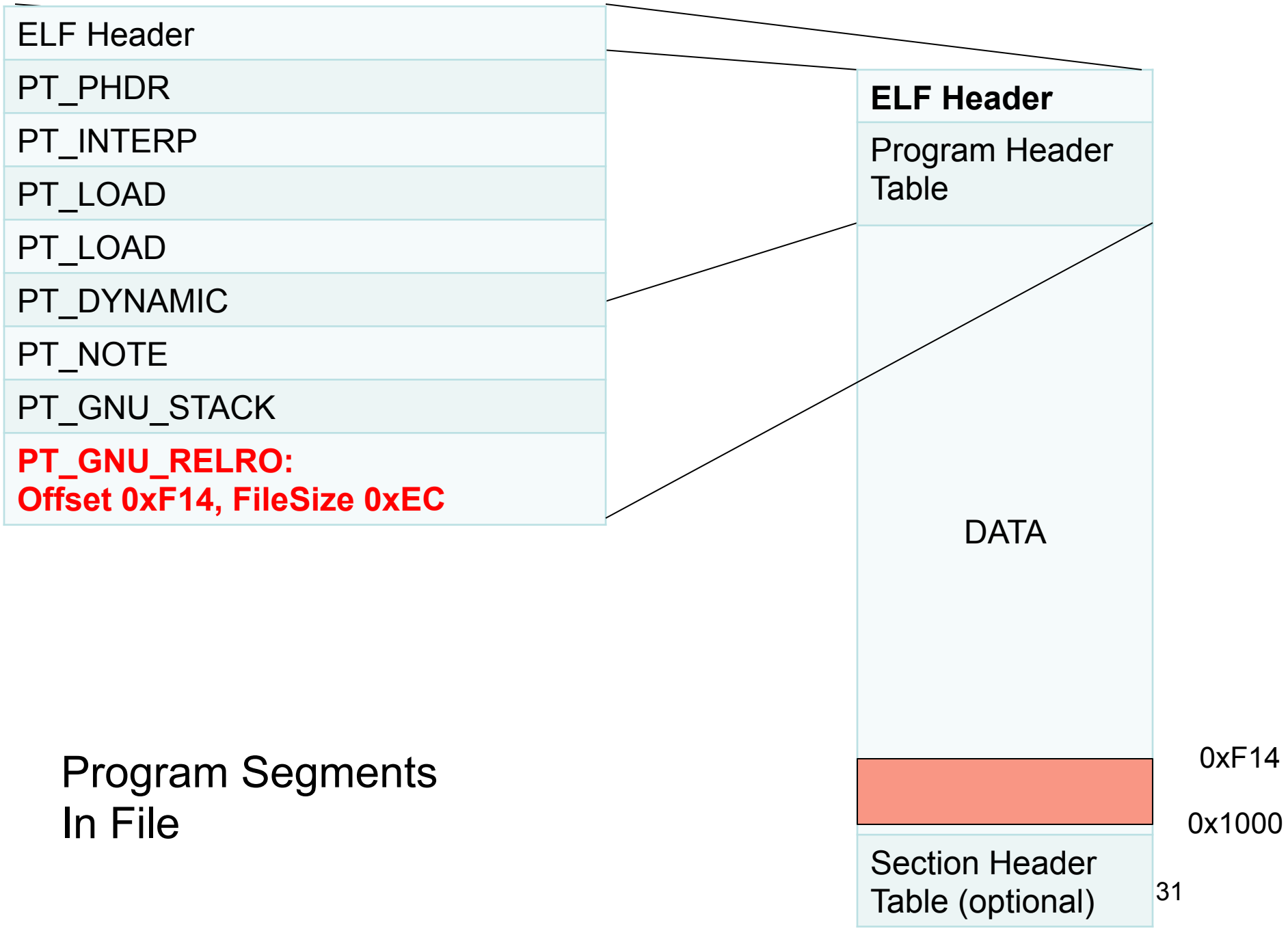

Program Segments In File




Program Segments In File



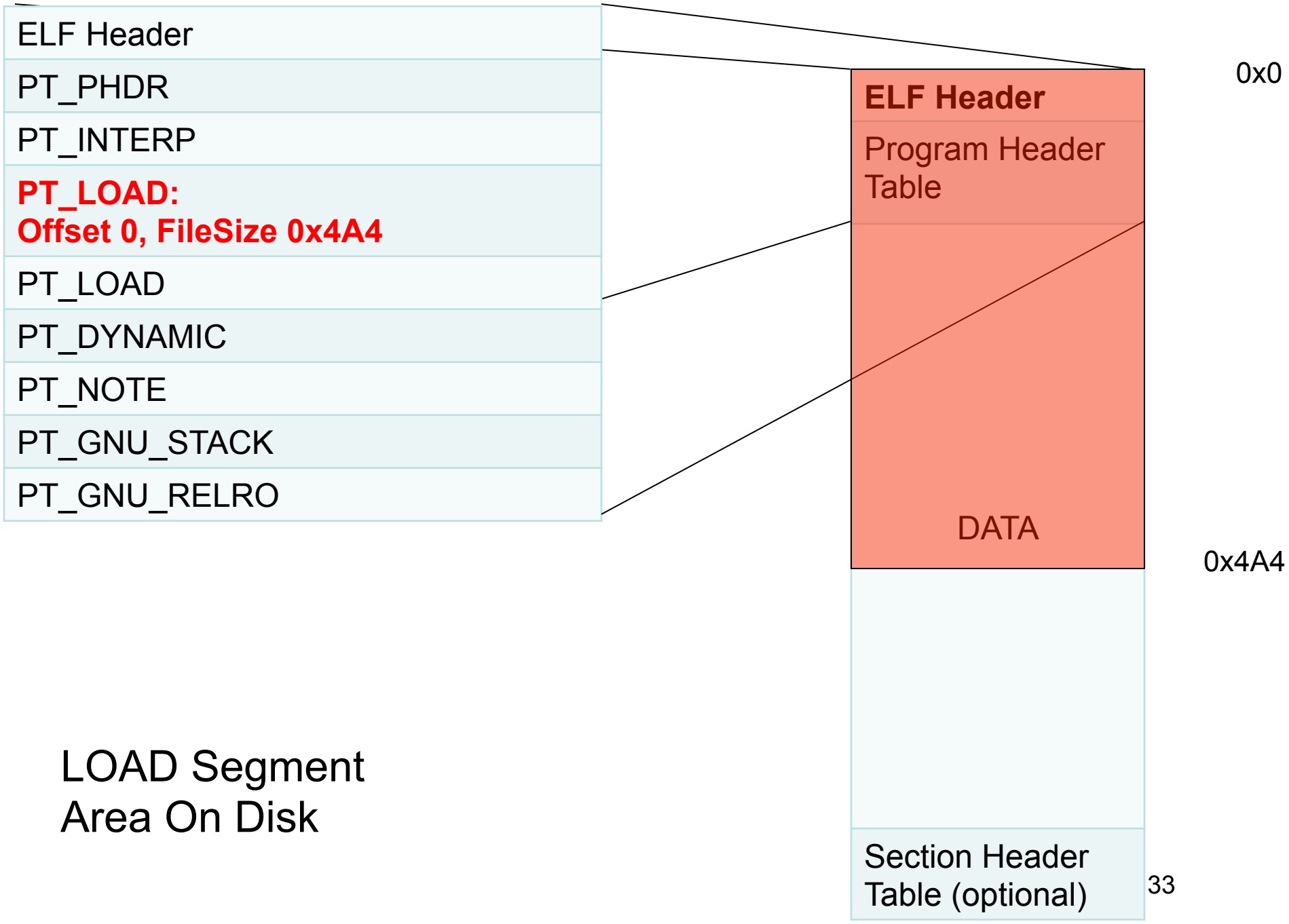

Program Segments
In File



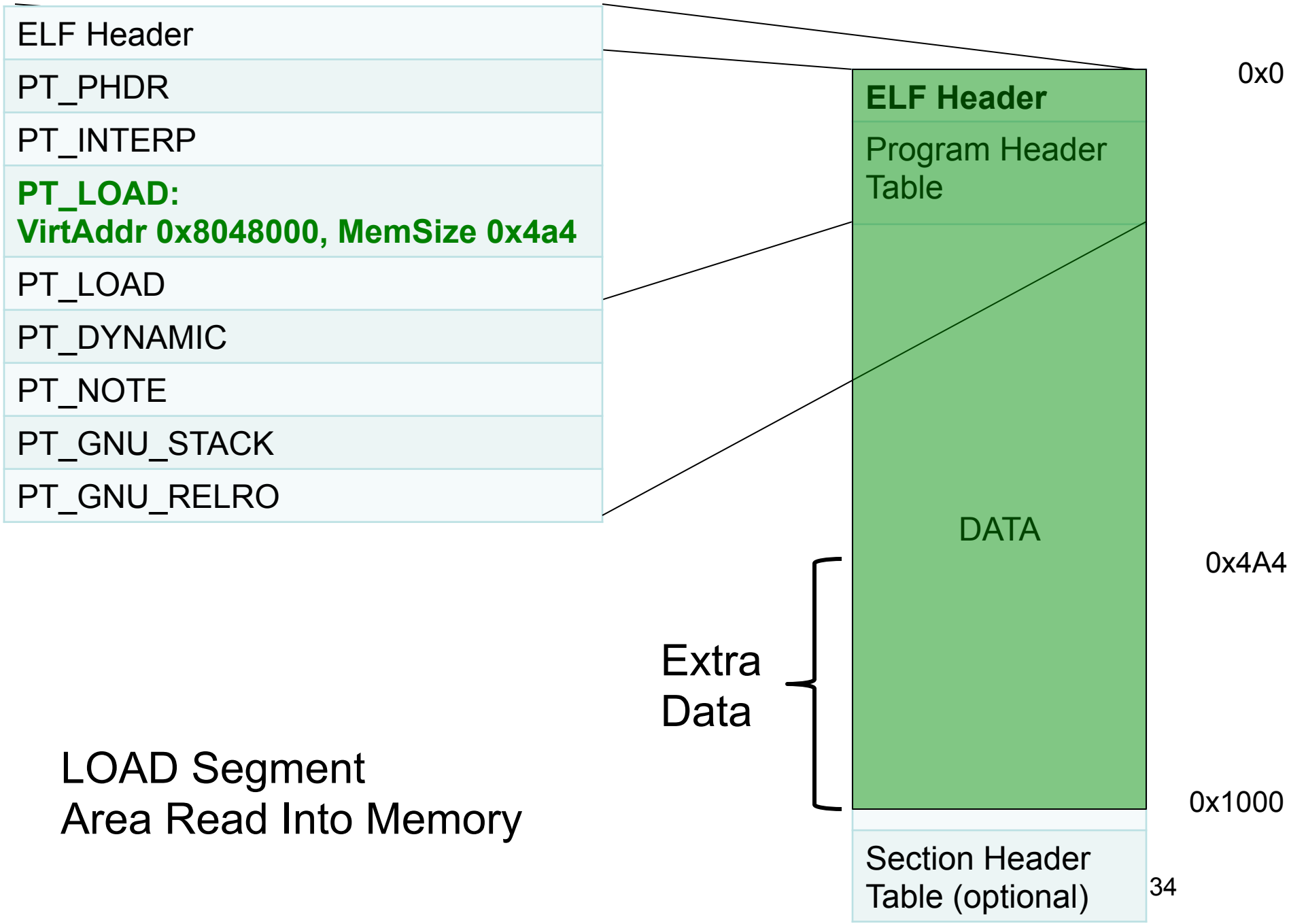

Program Segments
In File

# Mapping into memory

- The dynamic linker reads data from the ELF file in chunks 0x1000 bytes large (or possibly whatever size the PT_LOAD alignment is set to, but I've only ever seen 0x1000, and I don't want to look at the source code.)
- This leads to some interesting effects in terms of "padding" that occurs before or after the data that a segment actually specifies.
- In all cases, the loader reads the 0x1000 chunk such that the specified file offset will still map to the specified virtual address.

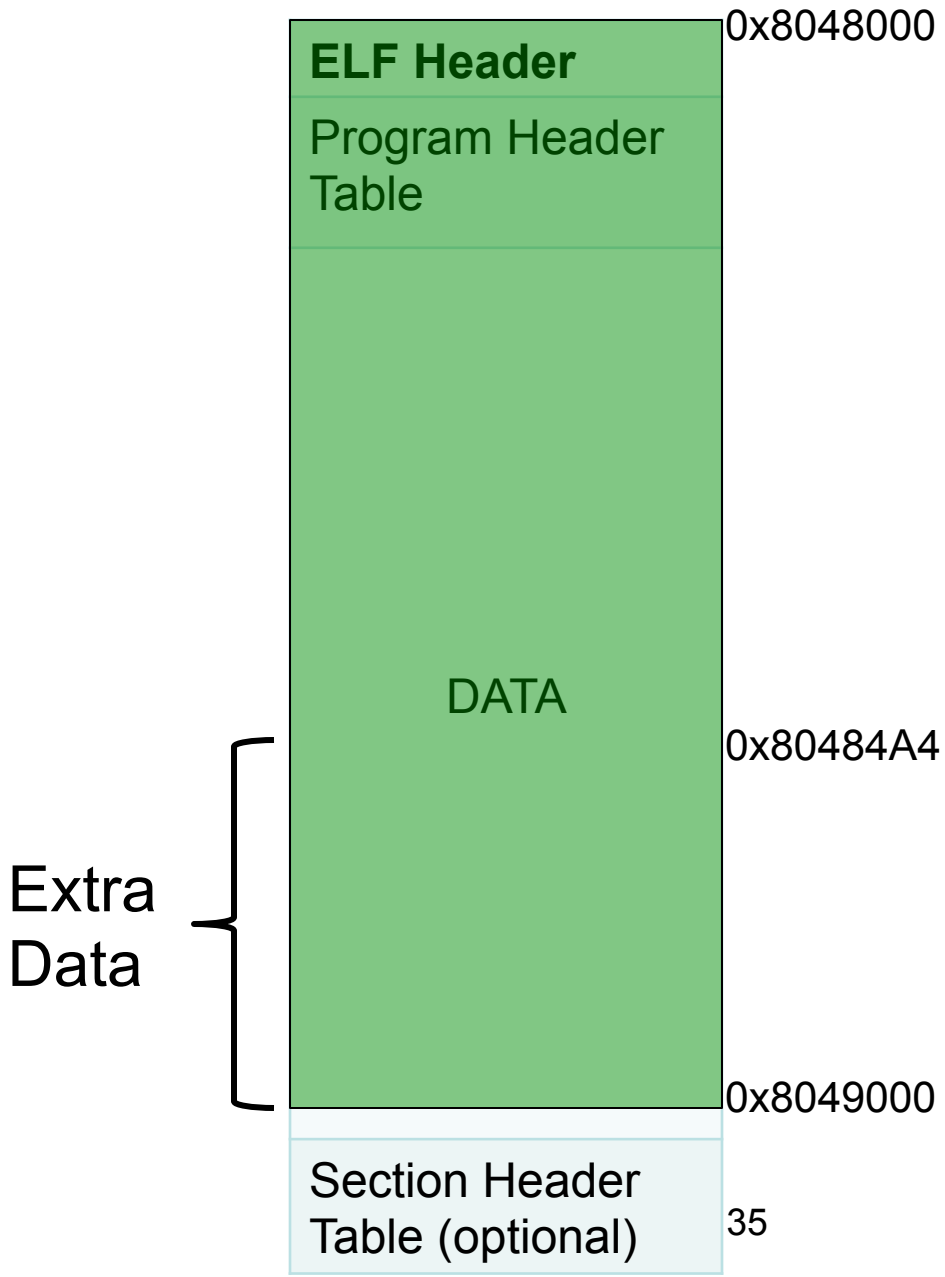


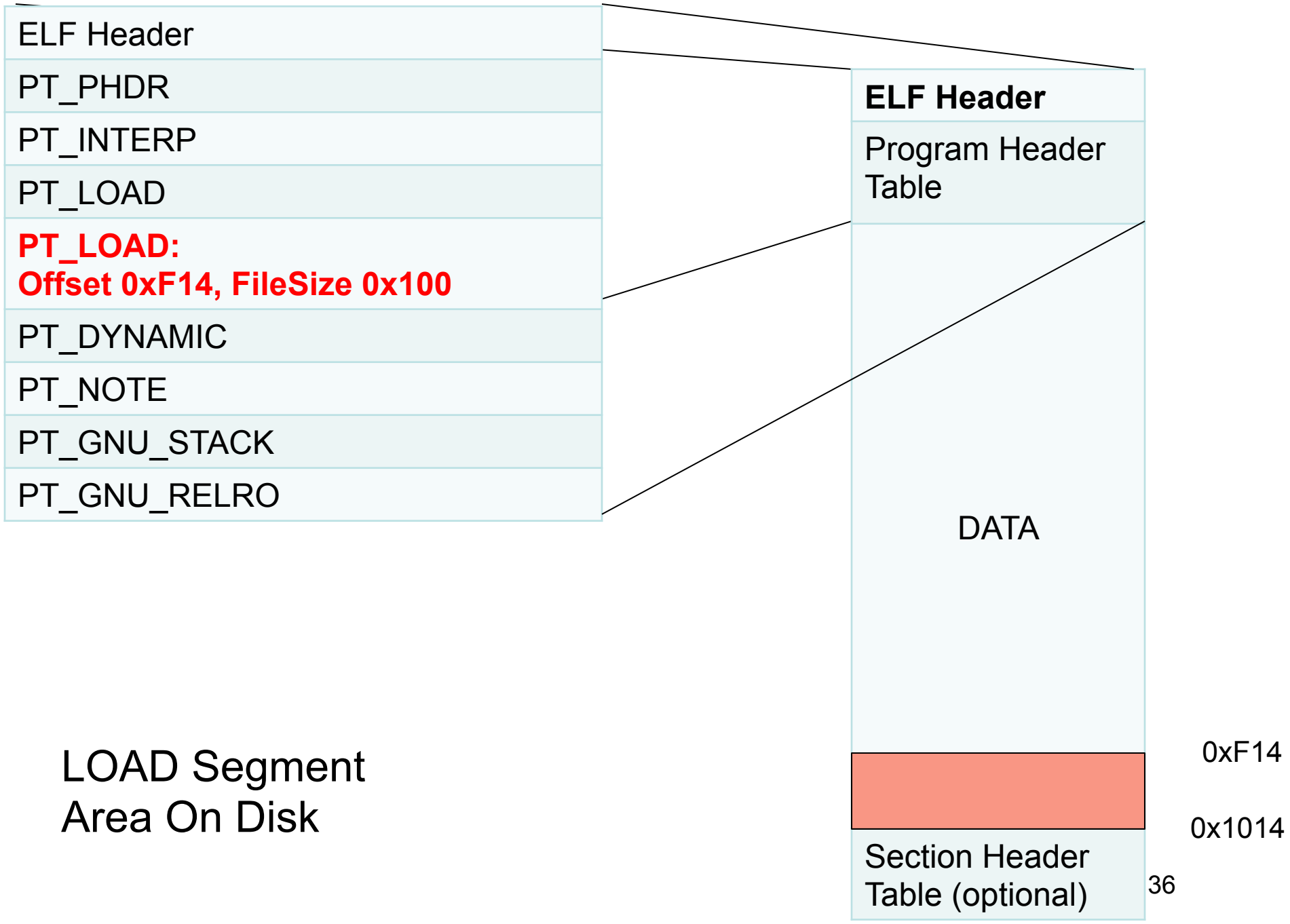

LOAD Segment
Area On Disk


ELF Header
PT_PHDR
PT_INTERP
PT_LOAD:
VirtAddr 0x8048000, MemSize 0x4a4
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_STACK
PT_GNU_RELRO
ELF Header
Program Header Table
DATA
Extra Data
Section Header Table (optional)
0x0
0x4A4
0x1000
34
LOAD Segment
Area Read Into Memory

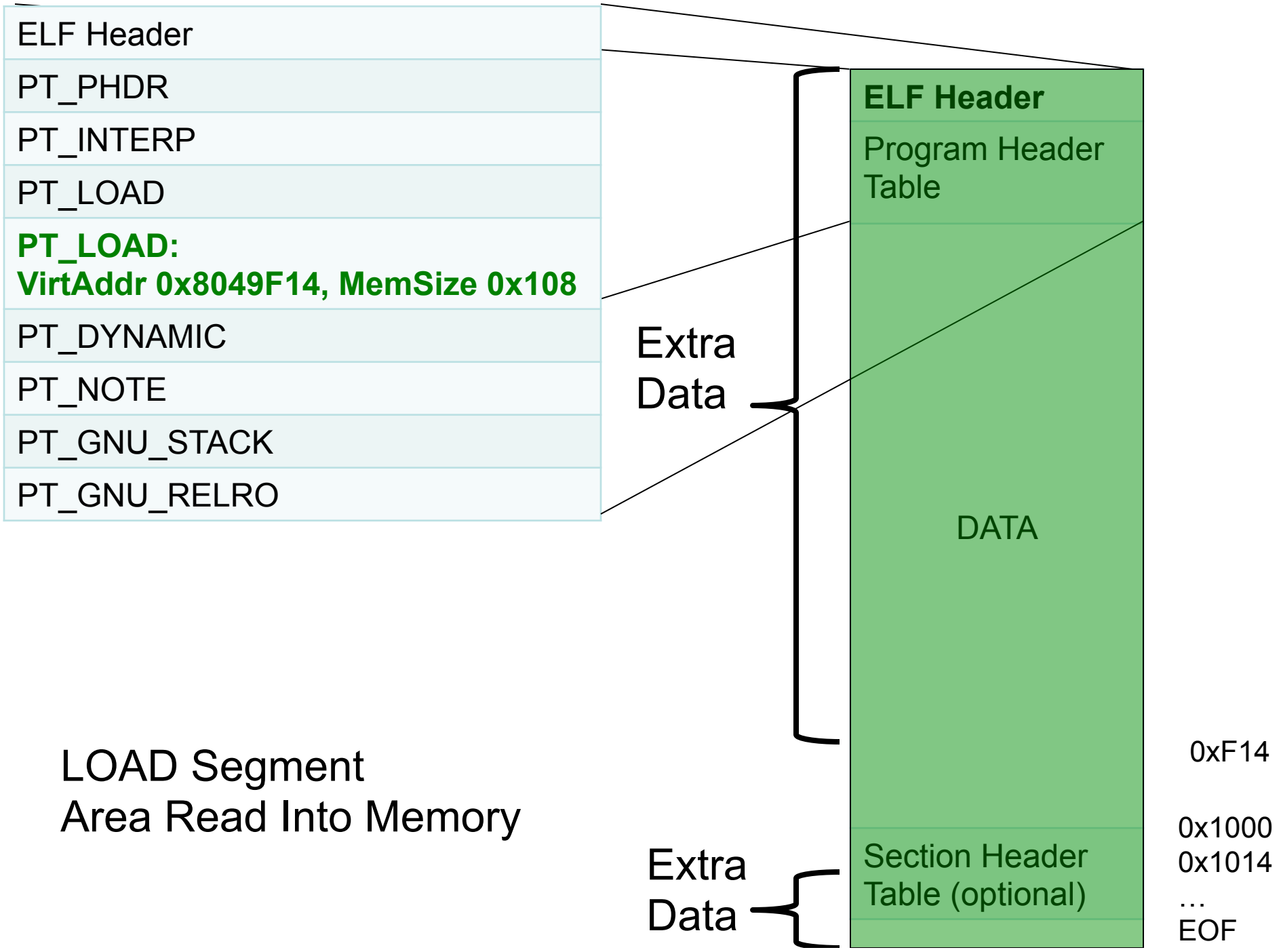
| |
|---|
| ELF Header |
| PT_PHDR |
| PT_INTERP |
| **PT_LOAD:**<br>**VirtAddr 0x8048000, MemSize 0x4a4** |
| PT_LOAD |
| PT_DYNAMIC |
| PT_NOTE |
| PT_GNU_STACK |
| PT_GNU_RELRO |

LOAD Segment
Area In Virtual Memory





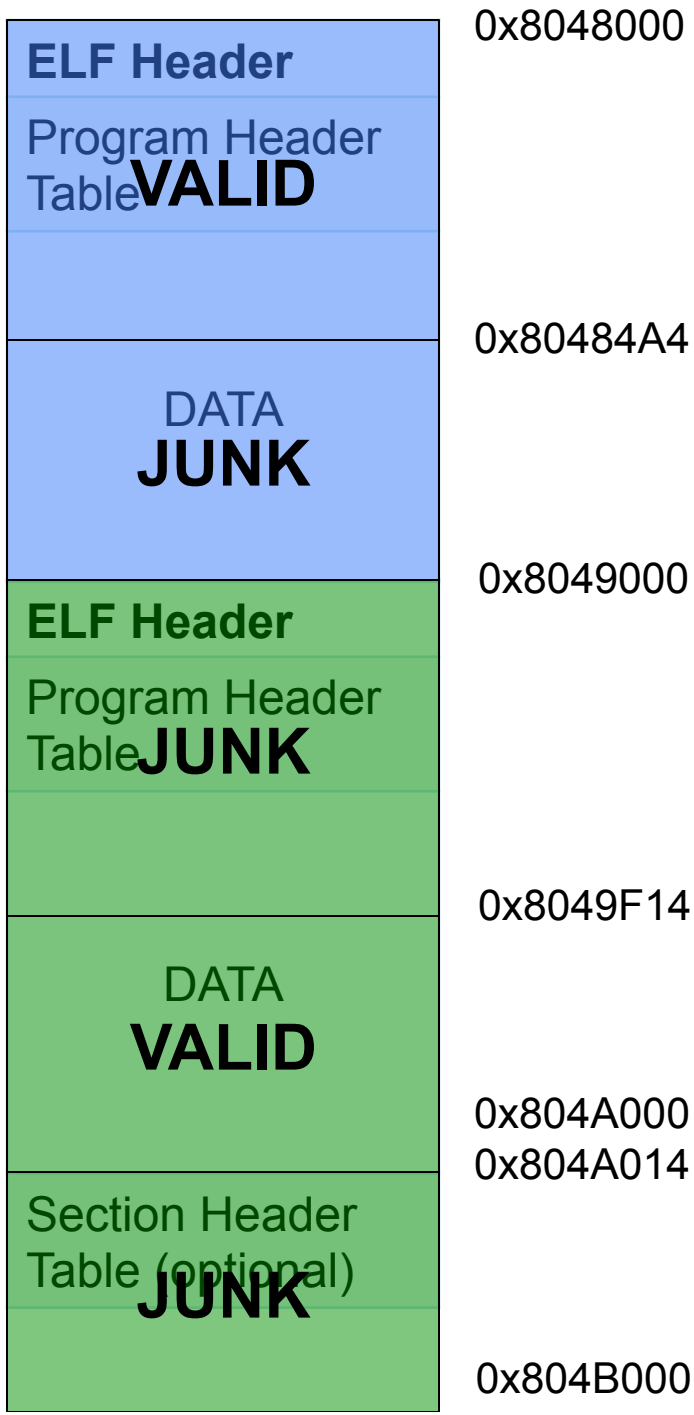

LOAD Segment
Area On Disk



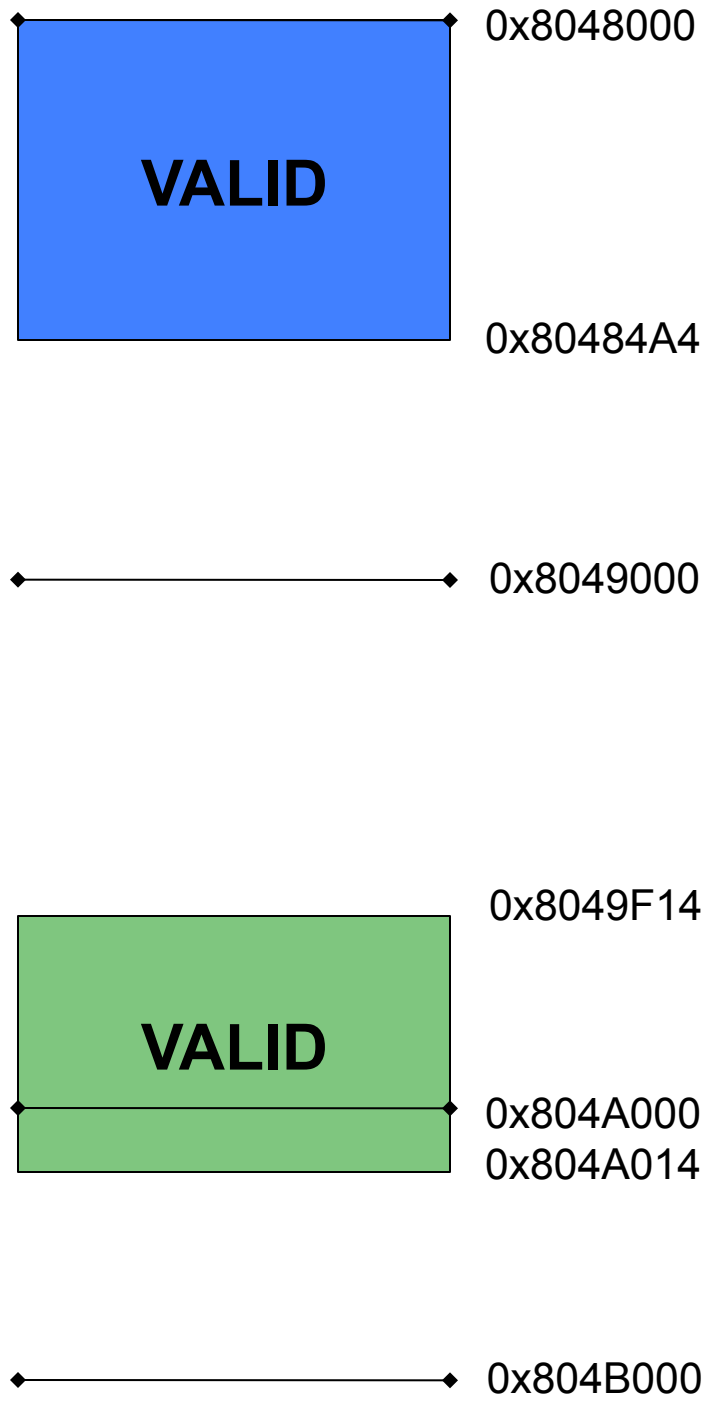

LOAD Segment
Area Read Into Memory

| |
|---|
| ELF Header |
| PT_PHDR |
| PT_INTERP |
| **PT_LOAD:**<br>**VirtAddr 0x8048000, MemSize 0x4a4** |
| **PT_LOAD:**<br>**VirtAddr 0x8049F14, MemSize 0x108** |
| PT_DYNAMIC |
| PT_NOTE |
| PT_GNU_STACK |
| PT_GNU_RELRO |

LOAD Segment
Area In Virtual Memory
**NOTE: I CHANGED SCALE!**



| |
|---|
| ELF Header |
| PT_PHDR |
| PT_INTERP |
| **PT_LOAD:**<br>**VirtAddr 0x8048000, MemSize 0x4a4** |
| **PT_LOAD:**<br>**VirtAddr 0x8049F14, MemSize 0x108** |
| PT_DYNAMIC |
| PT_NOTE |
| PT_GNU_STACK |
| PT_GNU_RELRO |




LOAD Segment
Area In Virtual Memory
***Intention***

**P.S. .BSS :)**
**FileSize = 0x100**
**MemSize = 0x108**

ELF Header

**PT_PHDR**

**PT_INTERP**

**PT_LOAD**

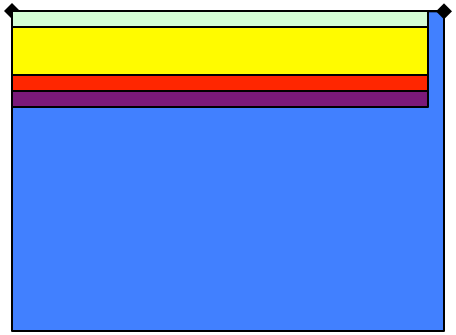
**PT_LOAD**

**PT_DYNAMIC**

**PT_NOTE**

**PT_GNU_STACK (N/A)**

**PT_GNU_RELRO**



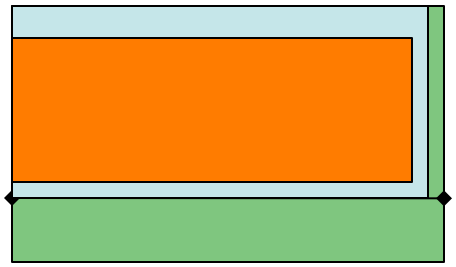
0x8048000

0x80484A4

← **Needs an old school Apple logo :) (but not right order)**

0x8049000

0x8049F14



0x804A000

0x804A014

**P.S. .BSS :)**
**FileSize = 0x100**
**MemSize = 0x108**

0x804B000

# LOAD Segment Area In Virtual Memory All Program Headers / Segments Accounted For

# How do I know…

- That the loader keeps reading after 0x4a4 for instance? Couldn't it just write that much data and then the rest could be whatever happened to be on the page which it was mapped to?
- It *could*, but it doesn't. You need proof pudding! If it's reading the data past 0x4A4, then the data at offset 0x4A5… should be in virtual memory at 0x80484A5…
- hexdump -C -s 0x4A5 -n 10 hello
- gdb ./hello
- b main
- r
- x/10xb 0x80484A5

# Haha, don't you hate zeros?

- How about the other way. If we believe that the loader is reading unnecessary data before the 0xF14 offset for the second load section, then 0x8049000 should be the same thing as offset 0 into the file.
- hexdump -C -n 10 hello
- gdb ./hello
- b main
- r
- x/10bx 0x8049000

# POP QUIZ!

- (Ask multiple students.)
- a) Name 1 way that PE sections are similar to ELF segments.
- b) Name 1 way they differ.

# Section Header

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

# Section Headers 2

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;       /* Section virtual addr at execution */
  Elf32_Off     sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section information */
  Elf32_Word    sh_addralign;   /* Section alignment */
  Elf32_Word    sh_entsize;    /* Entry size if section holds table */
} Elf32_Shdr;
```

# Section Headers 3

- **sh_name** is an offset in bytes into the string table which points to the name of the section. There is a null character at offset 0 in the string table, so anything with 0 for this value has no name. Reminder: the string table is found by consulting the e_shstrndx in the ELF Header which specifies an index into the section header table.
- As with the Program Header, the section header utilizes a type field, **sh_type**, and a bunch of different types which can specify vastly different interpretations for the section header.
- Ponder for a moment the parallels and perpendicularity of PE and ELF wrt sections and section typing. How do you specify a section type in PE?

# Section Headers 4

types for sh_type

- **SHT_PROGBITS** is a sort of the catch-all for anything which is valid but doesn't have some other special predefined type
- **SHT_STRTAB** is for string tables
- **SHT_DYNAMIC** is for dynamic linking information
- **SHT_NOBITS** is for things which take no space in the file but do take space in memory (like .bss)

# Section Headers 5

types for sh_type

- SHT_NULL is not used, and therefore other members of the section header are undefined. The first section header is always of this type.
- SHT_SYMTAB and SHT_DYNSYM are symbol tables used for linking or dynamic linking
- SHT_RELA and SHT_REL are for two different types of relocations talked about later.
- SHT_HASH is a symbol table hash

# Section Headers 6

- **sh_flags** can have the values SHF_WRITE = 0x1, SHF_ALLOC = 0x2, SHF_EXECINSTR = 0x4, or any number of processor-specific things as long as the MSB is set (which can be checked by ANDing with SHF_MASKPROC). Of these, SHF_ALLOC probably needs a little more explaining. SHF_ALLOC declares whether or not this section is going to occupy memory during program execution. The .debug* or .shstrtab are examples of sections that don't set this bit (and also set the sh_addr to 0 as was previously described as a way to indicate it won't reside in memory.)
- **sh_addr** is the virtual address where this section starts in memory. It's set to 0 if the section won't reside in memory.
- **sh_offset** is the file offset to the start of this data. This is still set for things with type SHT_NOBITS as it is the "conceptual" offset ;)

# Section Headers 7

- **sh_size** is the size of the section in bytes

**Figure 4-12:** `sh_link` **and** `sh_info` **Interpretation**

| sh_type | sh_link | sh_info |
|---|---|---|
| SHT_DYNAMIC | The section header index of the string table used by entries in the section. | 0 |
| SHT_HASH | The section header index of the symbol table to which the hash table applies. | 0 |
| SHT_REL<br>SHT_RELA | The section header index of the associated symbol table. | The section header index of the section to which the relocation applies. |
| SHT_SYMTAB<br>SHT_DYNSYM | The section header index of the associated string table. | One greater than the symbol table index of the last local symbol (binding STB_LOCAL). |
| other | SHN_UNDEF | 0 |

# Section Headers 8

- **sh_addralign** is the alignment constraint sh_addr if any (there is no constraint if it is set to 0 or 1). sh_addr mod sh_addralign must be 0, and sh_addralign must be an integral power of 2.
- Some sections such as a symbol table have an array of fixed-size fields. For such sections **sh_entsize** is the size in bytes of each entry. For sections which aren't organized this way, this field is 0.

# Viewing section header: readelf -S

```
user@ubuntu:~/code/hello$ readelf -S hello
There are 29 section headers, starting at offset 0x1170:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481ac 0001ac 000050 10   A  6   1  4
  [ 6] .dynstr           STRTAB          080481fc 0001fc 00004a 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048246 000246 00000a 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048250 000250 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048270 000270 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048278 000278 000018 08   A  5  12  4
  [11] .init             PROGBITS        08048290 000290 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        080482c0 0002c0 000040 04  AX  0   0  4
  [13] .text             PROGBITS        08048300 000300 00016c 00  AX  0   0 16
  [14] .fini             PROGBITS        0804846c 00046c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048488 000488 000015 00   A  0   0  4
  [16] .eh_frame         PROGBITS        080484a0 0004a0 000004 00   A  0   0  4
  [17] .ctors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        08049f1c 000f1c 000008 00  WA  0   0  4
  [19] .jcr              PROGBITS        08049f24 000f24 000004 00  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049f28 000f28 0000c8 08  WA  6   0  4
  [21] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        08049ff4 000ff4 000018 04  WA  0   0  4
  [23] .data             PROGBITS        0804a00c 00100c 000008 00  WA  0   0  4
  [24] .bss              NOBITS          0804a014 001014 000008 00  WA  0   0  4
  [25] .comment          PROGBITS        00000000 001014 00006c 01  MS  0   0  1
  [26] .shstrtab         STRTAB          00000000 001080 0000ee 00      0   0  1
  [27] .symtab           SYMTAB          00000000 0015f8 000400 10     28  44  4
  [28] .strtab           STRTAB          00000000 0019f8 0001fb 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

# POP QUIZ!

- (Ask multiple students.)
- a) Name 1 way that PE sections are similar to ELF sections.
- b) Name 1 way they differ.

# "Special Sections"

awwwww, idnt dat *special*

**Figure 4-13: Special Sections**

| Name | Type | Attributes |
|---|---|---|
| .bss | SHT_NOBITS | SHF_ALLOC + SHF_WRITE |
| .comment | SHT_PROGBITS | none |
| .data | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .data1 | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .debug | SHT_PROGBITS | none |
| .dynamic | SHT_DYNAMIC | see below |
| .dynstr | SHT_STRTAB | SHF_ALLOC |
| .dynsym | SHT_DYNSYM | SHF_ALLOC |
| .fini | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .got | SHT_PROGBITS | see below |
| .hash | SHT_HASH | SHF_ALLOC |
| .init | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .interp | SHT_PROGBITS | see below |
| .line | SHT_PROGBITS | none |
| .note | SHT_NOTE | none |
| .plt | SHT_PROGBITS | see below |
| .rel*name* | SHT_REL | see below |
| .rela*name* | SHT_RELA | see below |
| .rodata | SHT_PROGBITS | SHF_ALLOC |
| .rodata1 | SHT_PROGBITS | SHF_ALLOC |
| .shstrtab | SHT_STRTAB | none |
| .strtab | SHT_STRTAB | see below |
| .symtab | SHT_SYMTAB | see below |
| .text | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |

# Special Sections 2

just so you know what the names roughly mean

- .init = code which is called to initialize a runtime environment (e.g. initializes the C/C++ runtime)
- .fini = the flip side of .init, code which should be run at termination
- .text = main body of program code
- .bss = uninitialized writable data, takes no space in the file
- .data = initialized writable data
- .rdata = initialized read-only data (e.g. strings)
- .debug = symbolic debugging information (e.g. structure definitions, local variable names, etc). Not loaded into memory image.
- .line / .debug_line = correspondences between asm and source code line numbers. GNU stuff calls this .debug_line
- .comment = version information
- .note = "Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. Sections of type SHT_NOTE and program header elements of type PT_NOTE can be used for this purpose." There's an entire section specifying the form of note information, but the notes don't effect execution at all.

# Special Sections 3

just so you know what the names roughly mean

- .interp = the string specifying the interpreter. The PT_INTERP program header points at this section
- .dynamic = dynamic linking information. The PT_DYNAMIC program header points at this section.
- .dynstr = string table for dynamic linking
- .dynsym = dynamic linking symbol table
- .hash = symbol hash table
- .symtab = non-dynamic linking symbol table
- .strtab = string table, most often for non-dynamic linked symbol names
- .shstrtab = section header string table (names of section headers like ".text", ".data", etc

# Special Sections 4

just so you know what the names roughly mean

- .got = Global Offset Table used to help out position independent code
- .plt = Procedure Linkage Table used for "delay-load" aka "dynamic" aka "lazy" linking/resolution of imported functions.
- .got.plt = chunk of GOT used to support the PLT
- .rel* or .rela* = relocation information
- .ctors/.dtors = constructors/destructors not necessarily C++ con/destructors. You can use "__attribute__ ((constructor));" on functions to make them constructors, and then they will run before main(). You can apply "__attribute__ ((destructor));" and the function will run just after main() exits.
- .jcr = java class registration
- .eh_frame = exception frame

# The notorious PLT and the resolution

- The PLT supports "lazy" linking to imported functions. This is basically the same as the "delay-load" linking for PE.

# dynamic linking

hello

hi

how you doing?

fine, thanks.

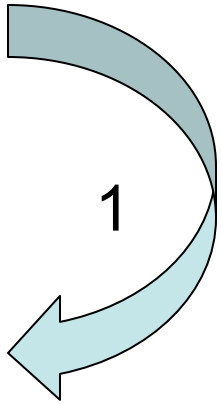
dynamic linker

libc
…
<puts>

.text
…
call <puts@plt>
…
call <puts@plt>

.plt
…
<puts@plt> jmp *<.got.plt+X>
<puts@plt+6> push $0x10
<puts@plt+11> jmp <dynamic linker>
…

.got.plt
…
<.got.plt+X> <puts@plt+6>
…



# dynamic linking

hello

hi

how you doing?

fine, thanks.

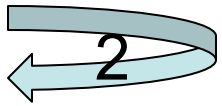
dynamic linker

libc
…
<puts>

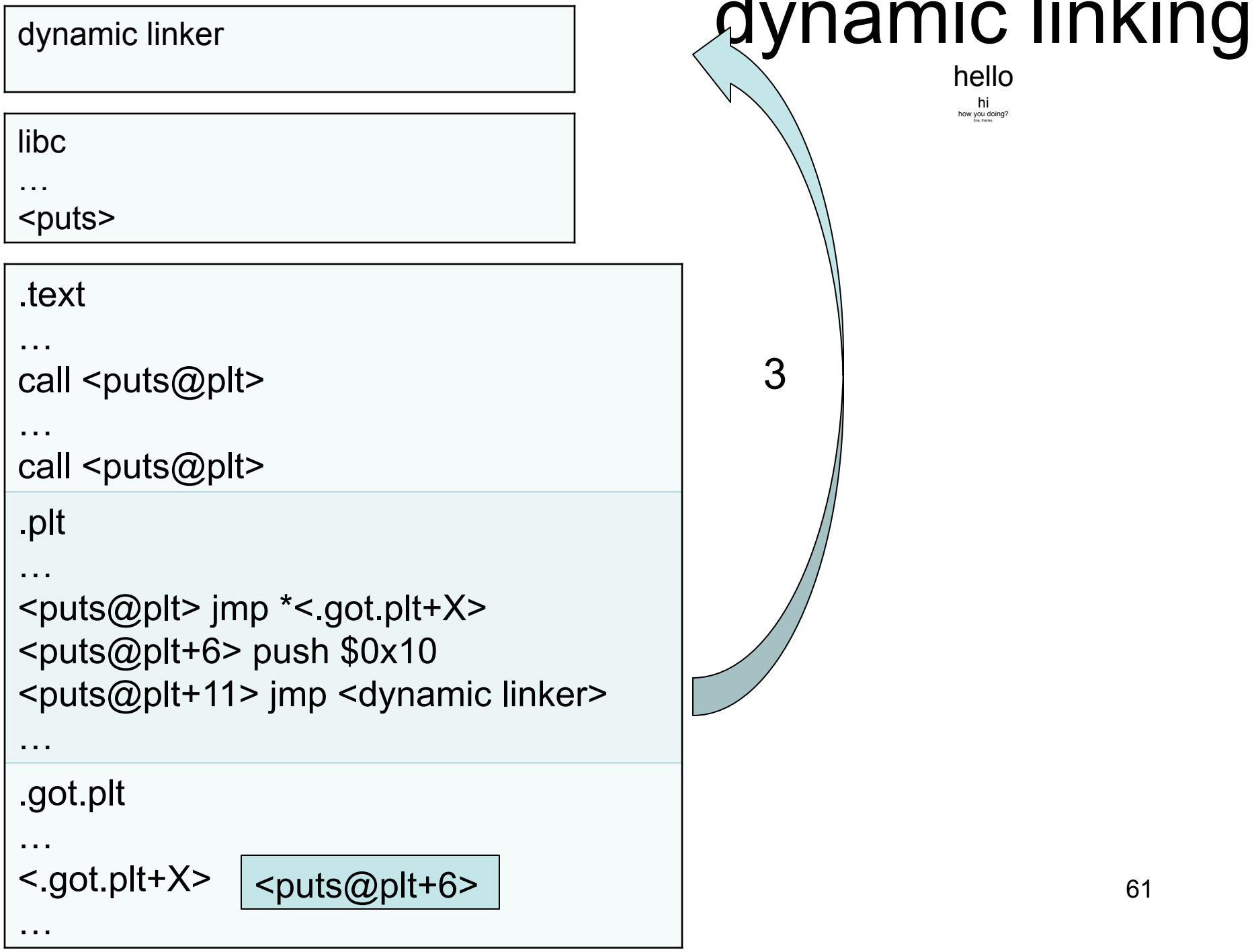
.text
…
call <puts@plt>
…
call <puts@plt>

.plt
…
<puts@plt> jmp *<.got.plt+X>
<puts@plt+6> push $0x10
<puts@plt+11> jmp <dynamic linker>
…

.got.plt
…
<.got.plt+X> <puts@plt+6>
…



# dynamic linking



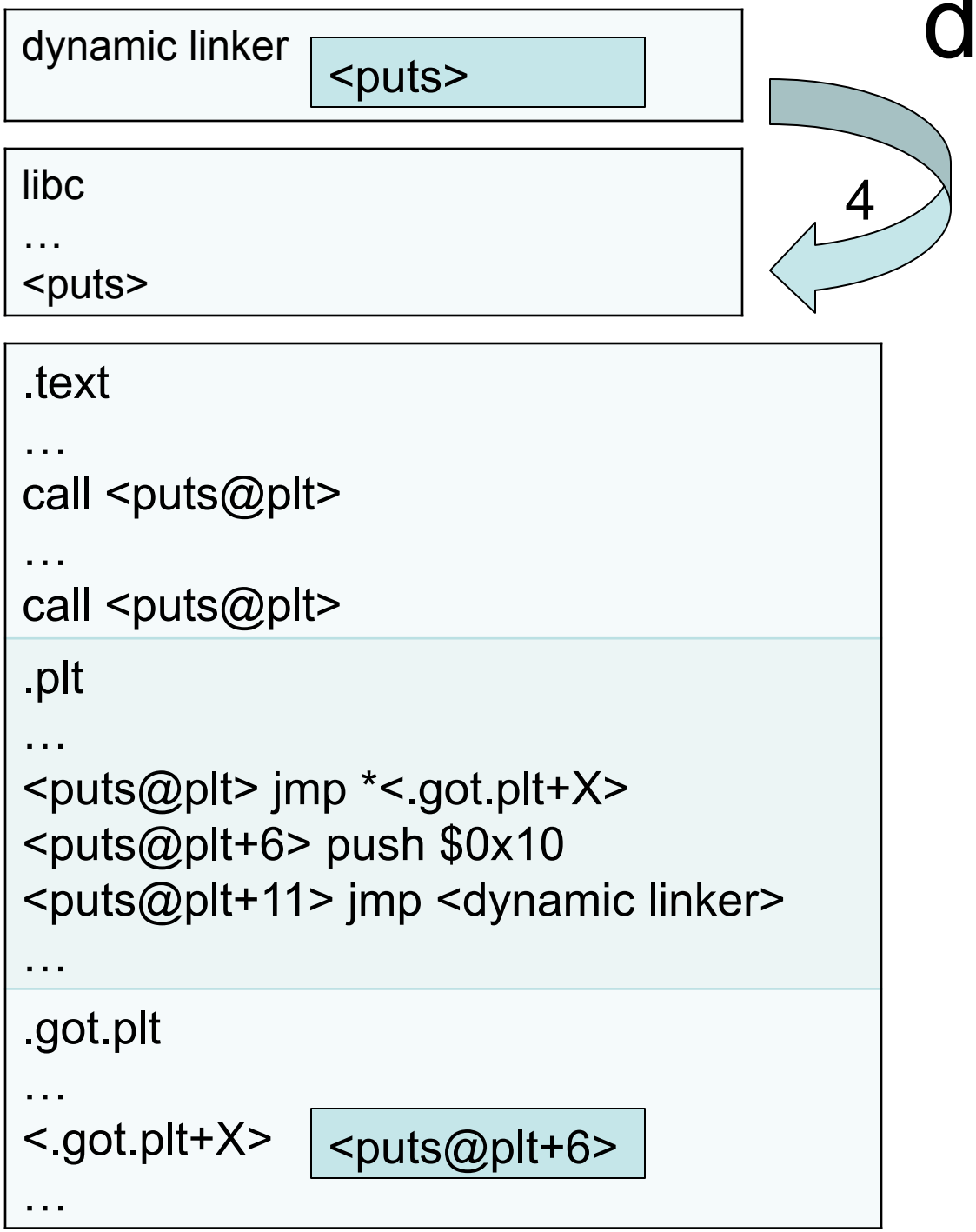
# dynamic linking

hello

hi

how you doing?

fine, thanks.



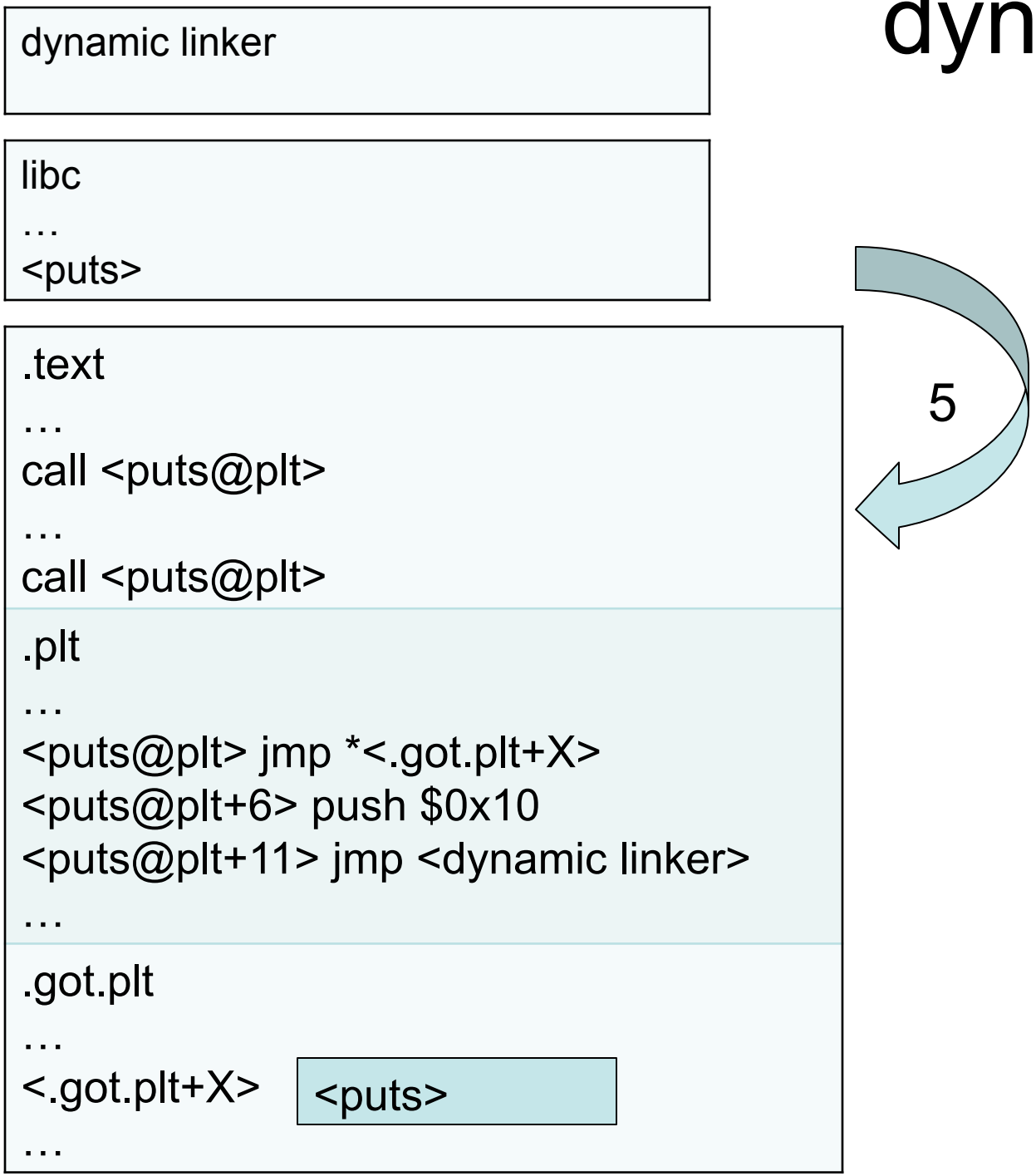
# dynamic linking

hello

hi

how you doing?

fine, thanks.



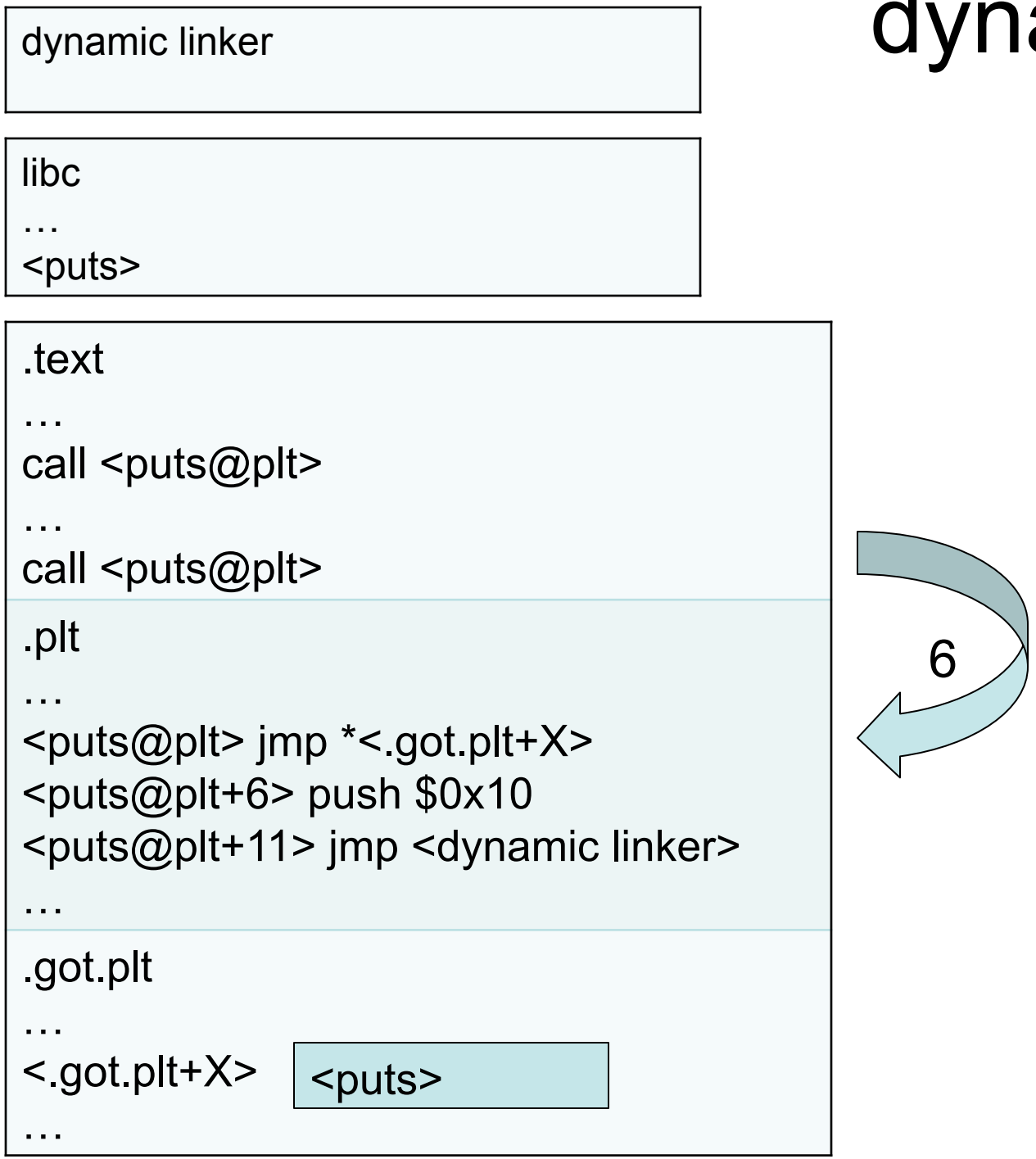
# dynamic linking

hello

hi

how you doing?

fine, thanks.



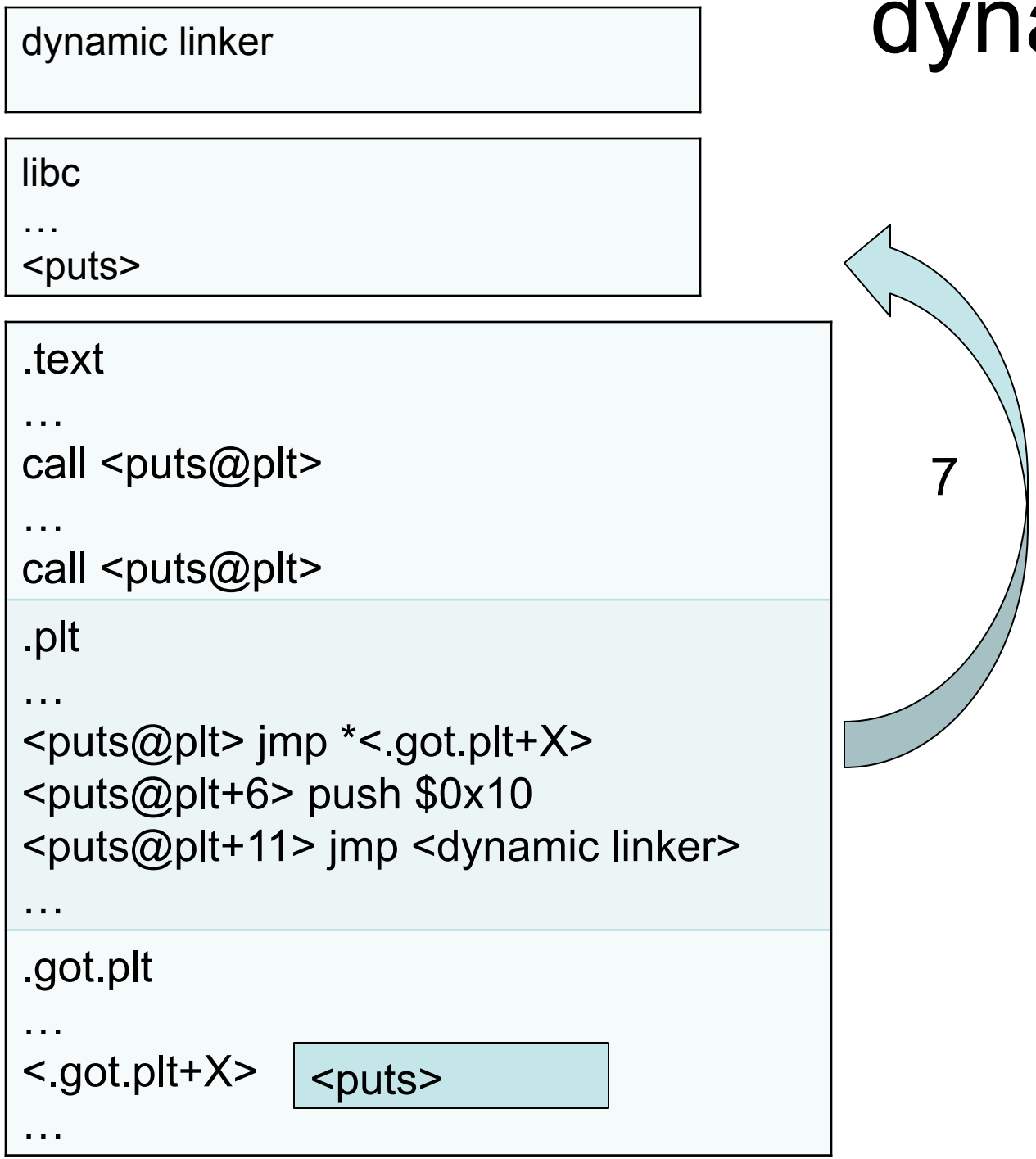
# dynamic linking

hello

hi

how you doing?

fine, thanks.



# Walk with me, won't you?

```
gdb ./hello2
(gdb) display/10i $eip
(gdb) b puts
(gdb) b *0x80483d0
(gdb) start
=> 0x80483b7 <main+3>:     and    $0xfffffff0,%esp
   0x80483ba <main+6>:     sub    $0x10,%esp
   0x80483bd <main+9>:     movl   $0x80484a0,(%esp)
   0x80483c4 <main+16>:    call   0x80482f0 <puts@plt>
   0x80483c9 <main+21>:    movl   $0x80484ad,(%esp)
   0x80483d0 <main+28>:    call   0x80482f0 <puts@plt>
   0x80483d5 <main+33>:    leave
   0x80483d6 <main+34>:    ret
(gdb) si 4
=> 0x80482f0 <puts@plt>:   jmp    *0x804a008
   0x80482f6 <puts@plt+6>: push   $0x10
   0x80482fb <puts@plt+11>:      jmp    0x80482c0
```

# A lovely day for a stroll

```
(gdb) x/x 0x804a008
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:      0x080482f6

[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[22] .got.plt          PROGBITS        08049ff4 000ff4 000018 04  WA  0   0  4

(gdb) si 3
=> 0x80482c0: pushl  0x8049ff8
   0x80482c6: jmp    *0x8049ffc

(gdb) x/x 0x8049ffc
0x8049ffc <_GLOBAL_OFFSET_TABLE_+8>:       0x00123ec0

(gdb) si 2
0x00123ec0 in ?? () from /lib/ld-linux.so.2
1: x/10i $eip
=> 0x123ec0:  push   %eax
   0x123ec1:  push   %ecx
   0x123ec2:  push   %edx
   0x123ec3:  mov    0x10(%esp),%edx
   0x123ec7:  mov    0xc(%esp),%eax
   0x123ecb:  call   0x11e080
```

**In the dynamic linker**

# The stillness of the air portends our doom, wouldn't you say?

```
(gdb) c
Continuing.

Breakpoint 2, 0x0018da96 in puts () from /lib/libc.so.6
1: x/10i $eip
=> 0x18da96 <puts+6>:  mov    %ebx,-0xc(%ebp)
   0x18da99 <puts+9>:  mov    0x8(%ebp),%eax
   0x18da9c <puts+12>: call   0x145b1f

(gdb) c
Continuing.
Hello world!

Breakpoint 3, 0x080483d0 in main ()
1: x/10i $eip
=> 0x80483d0 <main+28>:       call   0x80482f0 <puts@plt>
   0x80483d5 <main+33>:       leave
   0x80483d6 <main+34>:       ret
```

# Ominous

```
(gdb) si
0x080482f0 in puts@plt ()
1: x/10i $eip
=> 0x80482f0 <puts@plt>:    jmp    *0x804a008
   0x80482f6 <puts@plt+6>:  push   $0x10
   0x80482fb <puts@plt+11>: jmp    0x80482c0

(gdb) x/x 0x804a008
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:     0x0018da90

(gdb) x/10i 0x0018da90
   0x18da90 <puts>:  push   %ebp
   0x18da91 <puts+1>:mov    %esp,%ebp
```

TADA! The second time you go to call puts, the .got.plt entry is now filled in with the address of the function, rather than the address of some code in the dynamic linker

# ANY way you want it THAT'S the way you need it!

- If you set LD_BIND_NOW=1 in your environment variables, it will tell the dynamic linker to resolve all imports (PLT entries) before handing control to the program. (Remember, the dynamic linker is the "interpreter" which gets to run before the actual program.) This means you're forcing it to behave more like the Windows loader, which resolves all (quiz: except which?) imports at program start.
- export LD_BIND_NOW=1
- gdb ./hello
- proof pudding
- Now, proof pudding with special b*1 sauce (eat your heart out A1 sauce!)
- You can set an invalid breakpoint like "b *1" and then run and you will get control after things have been mapped into memory but before the dynamic linker has run.
- Also use "info proc" and "pmap <pid>" to prove dynamic linker is mapped into memory.



# PLT hooking

- Just like with IAT hooking on PE, we can do PLT hooking on ELF. But as with before, we first need a way to get into the memory address space.
- We will use the cheap LD_PRELOAD environment variable way of "shared object injection" (like DLL injection). With this variable set, the specified shared object will be loaded into memory earlier than any of the other imported shared objects, for every executable which is started

# Runtime Importing ELF

- Just as Windows has LoadLibrary() and GetProcAddress() which programmers can call to load libraries and run functions which aren't in the imports, so too does POSIX have dlopen () and dlsym().
- These functions can be abused by malware to obfuscate the used functions, etc, the same way the Windows ones can.
- FWIW glibc added on a dladdr() which takes a function pointer and tries to figure out the base address of the module it resides in, file path of the module, and symbol name of the function.
- RTFMan page for more info

# Thread Local Storage

- Much more about TLS:
- http://www.akkadia.org/drepper/tls.pdf
- Sorry, no callback functions like PE, thus making it fairly uninteresting. Instead, there's just an initialization blob that gets written into the location of the vars, so that they have their initialization values.

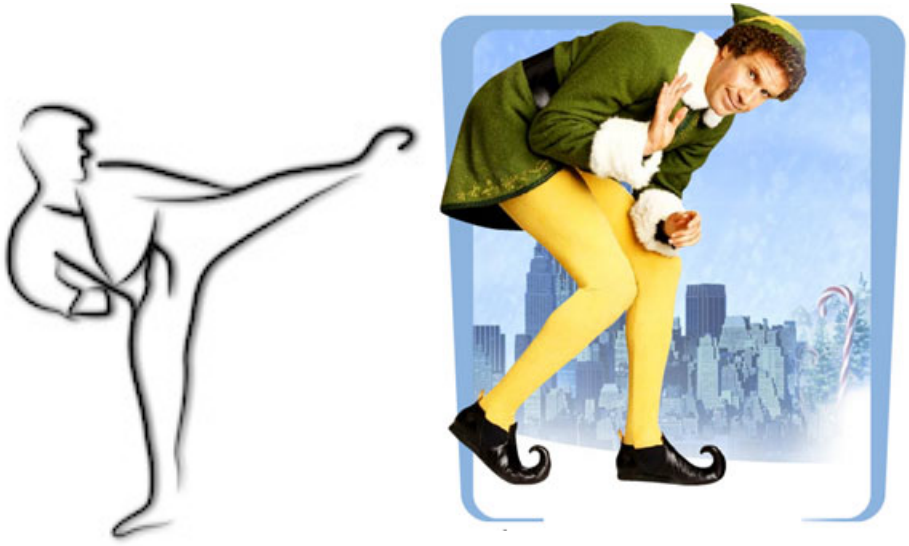
# TODO:

- Start talking about linking process again
- Then cover some compiler options and linker options

# Position Independent Code

- We saw the -fPIC option used in order to generate shared libraries for Linux. This generates position independent code, which is capable of being run no matter where it is loaded into memory. This is in contrast to normal "relocatable" code, which needs help from the OS loader in order to recalculate hardcoded offsets which are built into the assembly instructions.
- The Windows compiler cannot generate PIC code, all code requires fixups to be performed if the code is not loaded at it's "preferred" base address in memory.

# ELF Kickers: Kickers of ELF



- http://www.muppetlabs.com/~breadbox/software/elfkickers.html
- sstrip removes section headers
- objdump subsequently fails to disassemble the file
- gdb *used* to refuse to debug such a program, but it looks like they've fixed it

# Put your ELF on a diet with Diet Libc

- Diet Libc is a replacement for GNU Libc which tries to remove the bloat. (There are other such projects like uClibc (the u is micro), or tlibc (Tiny Libc))
- I just installed the dietlibc-dev ubuntu package
- diet gcc hello.c -o hello-dietlibc
- `711 hello-dietlibc-sstripped`
- `2484 hello-dietlibc`
- `4108 hello-sstripped`
- `5528 hello-stripped`
- `7155 hello`
- `8247 hello-ggdb`
- `616096 hello-static`