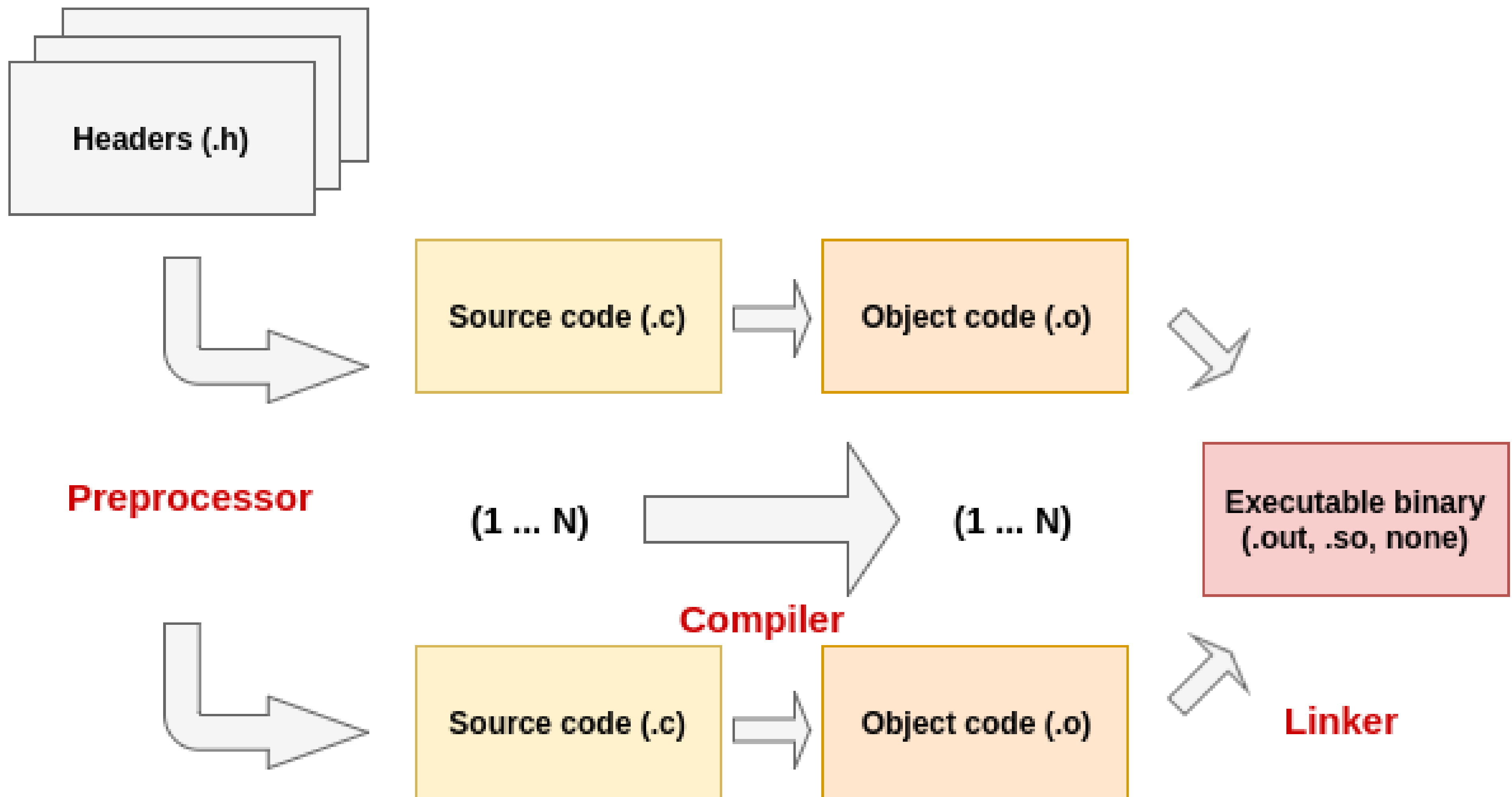
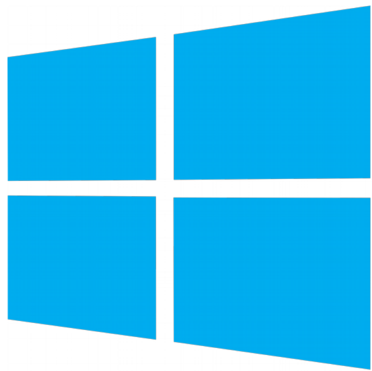
Reverse Engineering
Class 1

Executable Binaries

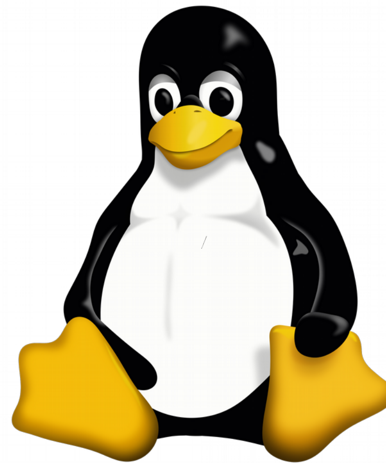
Compilation



Executable binaries



EXE (PE)



ELF

MACH-O

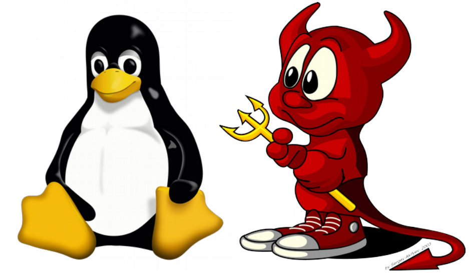
.NET assemblies

CLASSFILE

Executable binaries

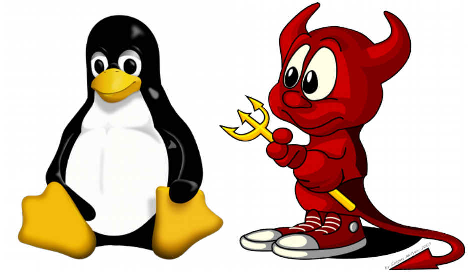
- Executable instructions
  - by the processor (x86, ARM, etc.)
  - by a virtual machine interpreter (JVM, CPython, etc.)
- Data
  - constants (embedded in instruction opcodes or not)
  - variables
- Debugging information

ELF



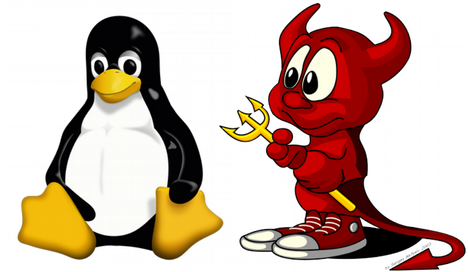
- Executable and Linkable Format (ELF)
  - Linux Foundation reference specification
  - ELF headers
    - <glibc>/elf/elf.h (glibc)
    - /usr/include/linux/elf.h (kernel)
    - man elf
  - Specify executable binaries, libraries (shared-objects), objects (.o), kernel modules, the kernel itself
  - Binary format. Why? Could it be different?

ELF



- ELF format is recognized by:
  - kernel
    - when loading modules and drivers
    - sys_execve when launching executable binaries
  - dynamic loader (ld-linux, glibc)
    - when loading shared libraries
  - bootloader
    - when loading the kernel in memory
  - compilers, linkers, assemblers, debuggers and other tools (objdump, readelf, etc.)

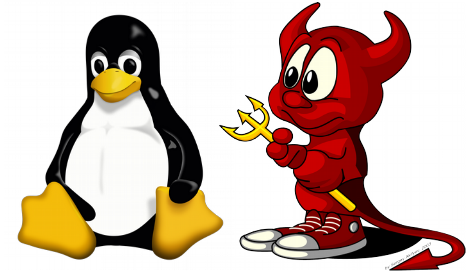
ELF



```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  .ELF............
00000010  03 00 3E 00 01 00 00 00 70 F0 02 00 00 00 00 00  ..>.....p.......
00000020  40 00 00 00 00 00 00 00 48 54 10 00 00 00 00 00  @.......HT......
00000030  00 00 00 00 40 00 38 00 09 00 40 00 1D 00 1C 00  ....@.8...@.....
00000040  06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000050  40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  @.......@.......
00000060  F8 01 00 00 00 00 00 00 F8 01 00 00 00 00 00 00  ................
00000070  08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00  ................
00000080  38 02 00 00 00 00 00 00 38 02 00 00 00 00 00 00  8.......8.......
00000090  38 02 00 00 00 00 00 00 1C 00 00 00 00 00 00 00  8...............
000000a0  1C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
000000b0  01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00  ................
000000c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000d0  54 75 0F 00 00 00 00 00 54 75 0F 00 00 00 00 00  Tu......Tu......
000000e0  00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00  .. .............
0000000f  A8 7E 0F 00 00 00 00 00 A8 7E 2F 00 00 00 00 00  .~.......~/.....
00000100  A8 7E 2F 00 00 00 00 00 C0 B6 00 00 00 00 00 00  .~/.............
00000110  90 10 01 00 00 00 00 00 00 20 00 00 00 00 00 00  ......... ......
00000120  02 00 00 00 06 00 00 00 80 A6 0F 00 00 00 00 00  ................
00000130  80 A6 2F 00 00 00 00 00 80 A6 2F 00 00 00 00 00  ../......./.....
00000140  E0 01 00 00 00 00 00 00 E0 01 00 00 00 00 00 00  ................
```

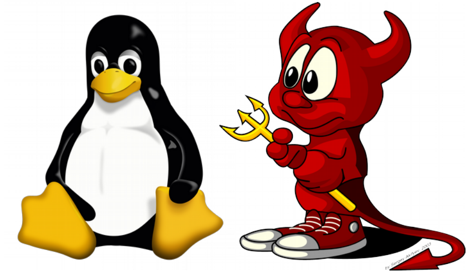
/bin/bash (Linux, x86_64)

ELF



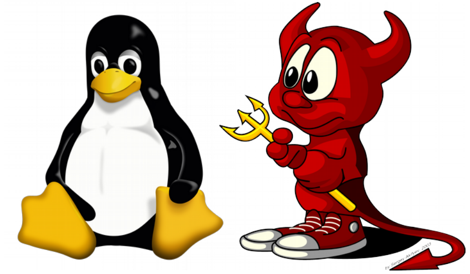
- File header
  - Magic number → “ELF” (0x7F, 0x45, 0x4C, 0x46)
  - 32 or 64 bits
  - Architecture → Linux supports multiple platforms
  - Endianness → Little-endian? Big-endian?
  - ELF version
  - ABI (Application Binary Interface) and ABI version
    - I.e. System V

ELF



- Type (relocatable object, executable binary, shared library)
- Entry point → first executable instruction
- Table offsets (in file)
  - Program Headers
  - Section Headers
- Table sizes (entry size, entries count)
- ELF header size

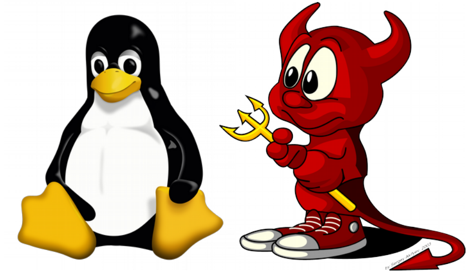
ELF



```
typedef struct
{
  unsigned char  e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf64_Half e_type;             /* Object file type */
  Elf64_Half e_machine;          /* Architecture */
  Elf64_Word e_version;          /* Object file version */
  Elf64_Addr e_entry;        /* Entry point virtual address */
  Elf64_Off  e_phoff;        /* Program header table file offset */
  Elf64_Off  e_shoff;        /* Section header table file offset */
  Elf64_Word e_flags;        /* Processor-specific flags */
  Elf64_Half e_ehsize;       /* ELF header size in bytes */
  Elf64_Half e_phentsize;        /* Program header table entry size */
  Elf64_Half e_phnum;        /* Program header table entry count */
  Elf64_Half e_shentsize;        /* Section header table entry size */
  Elf64_Half e_shnum;        /* Section header table entry count */
  Elf64_Half e_shstrndx;         /* Section header string table index */
} Elf64_Ehdr;
```

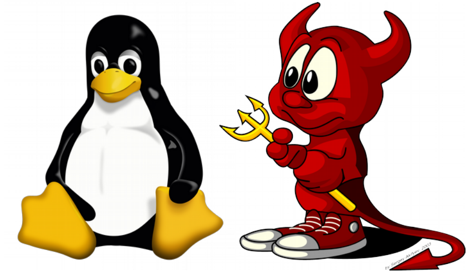
file header - elf.h (glibc)

ELF



How to parse an ELF header? (given a stream of bytes and a definition of the header structure)

ELF



How to parse an ELF header? (given a stream of bytes and a definition of the header structure)

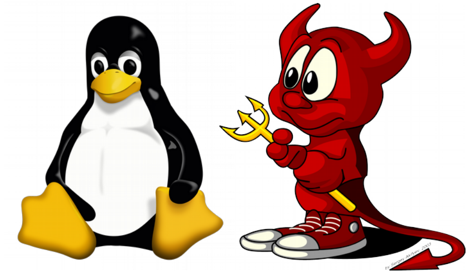
```
char* file_buffer = { 0x7F, 0x45, 0x4C, … };
Elf64_Ehdr *elf_header = (Elf64_Ehdr*)file_buffer;
elf_header->e_type;
```

CAST

This is what kernel, glibc, objdump, readelf and other tools do for fixed length structures.

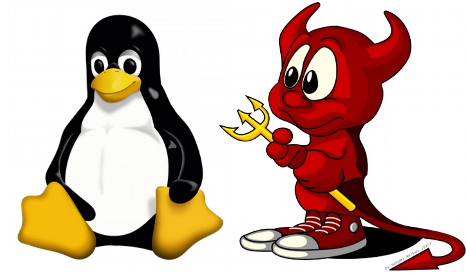
Ptr is incremented by sizeof(structure) to continue parsing.

ELF



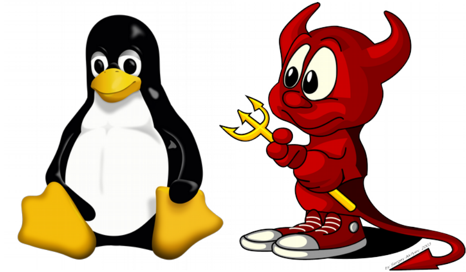
```
[martin@vmlinwork 1]$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400400
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6728 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

ELF



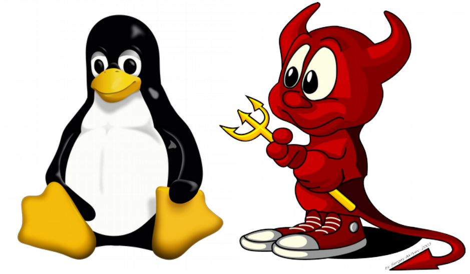
- **Program headers**
  - Table with information to prepare the binary for execution. Used in runtime too.
    - PT_LOAD → map of virtual memory segments to the executable binary:
      - offset in file
      - Virtual address + alignment
      - permissions (R, W, X)
      - file size
      - memory size
        - completed with 0s if greater than the size on file (i.e. uninitialized global variables)

ELF



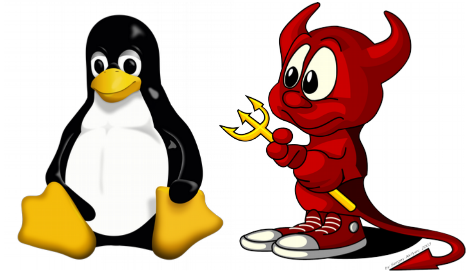
- **Program headers**
  - PT_LOAD
    - What does mapping virtual memory segments to a file mean?
    - If the file offset that we want to map were 0xdf8, what granularity do we have to map it to 0x600000?
    - What happens when segments can be written? Is the file written too?

ELF



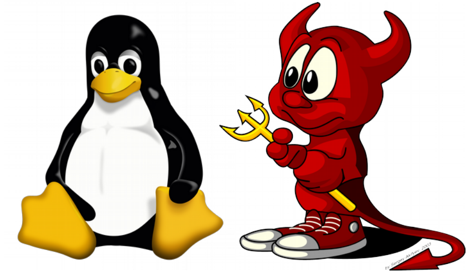
- **Program headers**
  - PT_LOAD
    - Virtual memory backed by a file. In case of page_fault, OS loads the data in memory from the file. Thus, the executable binary is not loaded to memory at once but in chunks as needed.
    - Page granularity (I.e. 4096 bytes). Thus, file is mapped from offset 0 to virtual address 0x600000.
    - Depends. In this case, file in HDD is not written.

ELF



- Who uses PT_LOAD information from an executable binary?

ELF

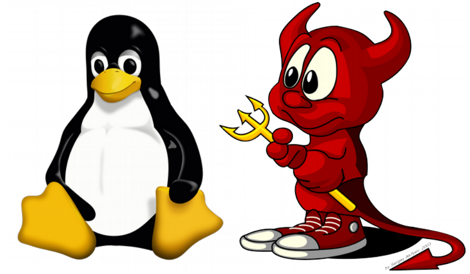


- Who uses PT_LOAD information from an executable binary?

Linux kernel: sys_execve maps process virtual address ranges to an executable binary (data, instructions).

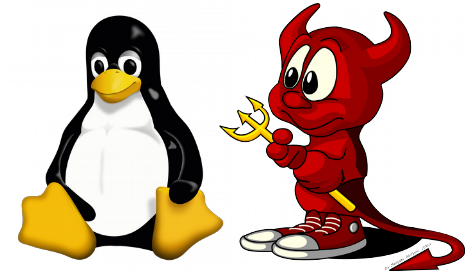
Dynamic loader: does the same for dynamic libraries.

ELF

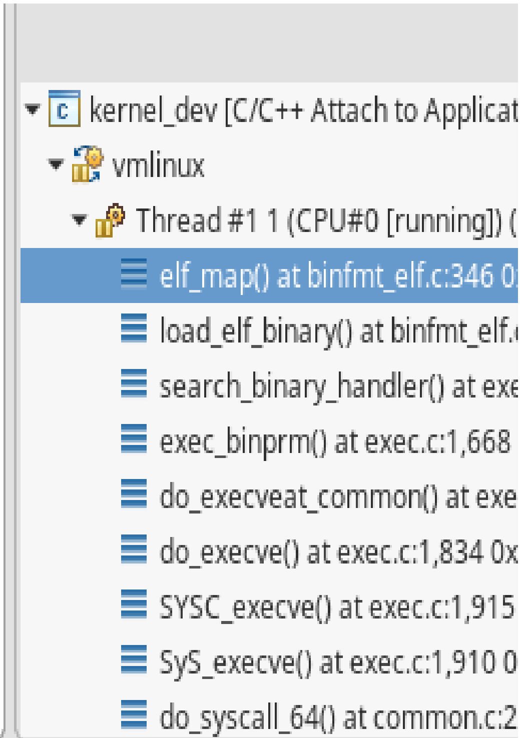


- Note: binaries and dynamic libraries may NOT specify to which virtual addresses have to be mapped
  - Position Independent Executables and Position Independent Code
  - Addresses are randomly chosen
  - In this case, PT_LOAD entry does not specify an absolute virtual address (just offsets, sizes, permissions, alignment, etc.)

ELF



```
atic unsigned long elf_map(struct file *filep, unsigned long addr,
		struct elf_phdr *eppnt, int prot, int type,
		unsigned long total_size)
{
	unsigned long map_addr;
	unsigned long size = eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr);
	unsigned long off = eppnt->p_offset - ELF_PAGEOFFSET(eppnt->p_vaddr);
	addr = ELF_PAGESTART(addr);
	size = ELF_PAGEALIGN(size);

	/* mmap() will return -EINVAL if given a zero size, but a
	 * segment with zero filesize is perfectly valid */
	if (!size)
		return addr;
```



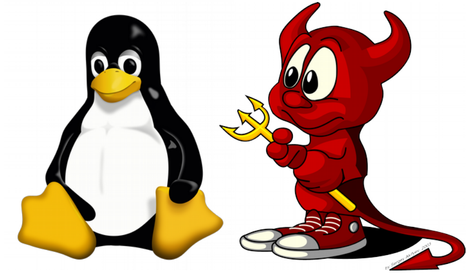
ole Tasks Problems Executables Debugger Console Memory Progress Search

ev [C/C++ Attach to Application] gdb (7.12.1)

```
0x8048000
```

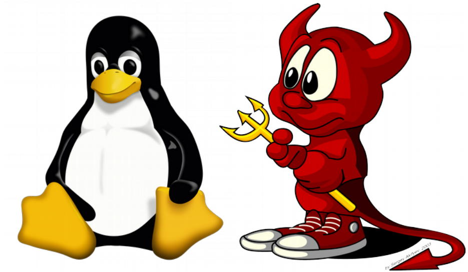
main_32 mapped in Linux kernel (x86_64) using PT_LOAD information – elf_map – fs/binfmt_elf.c. VirtAddress in PT_LOAD for executable segment is 0x08048000

ELF



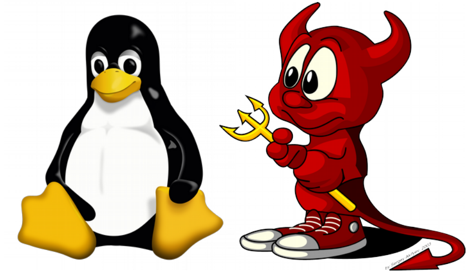
- PT_INTERP → binary interpreter (dynamic loader)
  - /lib64/ld-linux-x86-64.so.2
    - <glibc>/elf/ (source code)
  - Executes before the binary and loads libraries to which the binary dynamically links. It may not exist (self-contained binaries), but there is generally one.
- PT_DYNAMIC → information for the dynamic linker: in which virtual address is this information available? (according to PT_LOAD mapped from the file to a virtual RW segment)
- PT_PHDR → location of Program Headers table in virtual memory

ELF



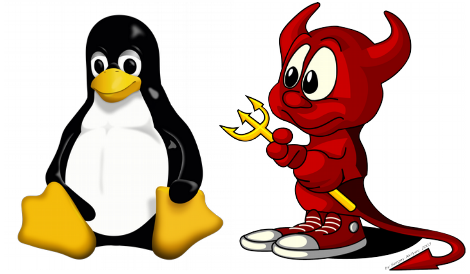
```
[martin@vmlinwork 1]$ readelf -l main

Elf file type is EXEC (Executable file)
Entry point 0x400400
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006e4 0x00000000000006e4  R E    200000
  LOAD           0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x000000000000021c 0x0000000000000238  RW     200000
  DYNAMIC        0x0000000000000e20 0x0000000000600e20 0x0000000000600e20
                 0x00000000000001d0 0x00000000000001d0  RW     8
```

ELF



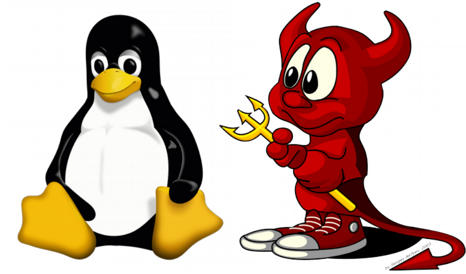
- **Section headers**
  - Map of executable binary sections in HDD (file). Each section is an "information unit"
    - Section name
    - Section type. In example:
      - Symbols table
      - Strings table
      - Relocations table
      - Dynamic symbols table
      - Hashes table
      - Dynamic linking table (referenced from Program Headers)

ELF



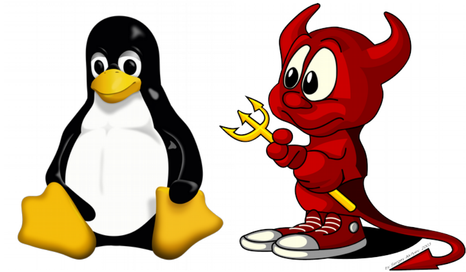
- Contains
  - Section offset in file
  - Section size in file (in memory if section were of NOBITS type)
  - Section virtual address
  - Alignment (to be loaded/mapped into the virtual address space; in HDD sections are contiguous)
  - Permissions (Read? Write? Execute?)
  - Entry size, if section were a table

ELF



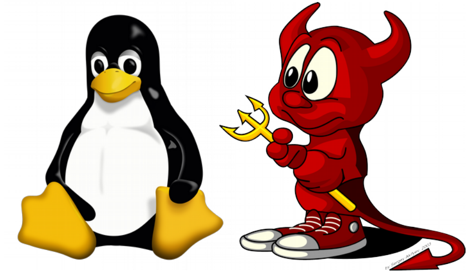
- This table (in file) is NOT mapped into the created process virtual address space
  - I.e.: readelf -S main_32
  - “There are 36 section headers, starting at offset 0x2700:”
  - However, PT_LOAD maps a maximum of 0x00124 file bytes starting at offset 0x000f00. That is up to offset 0x1024.
- However, data referenced from this table (sections themselves) may be mapped. Example: .text section. See Program Headers table and calculate file offsets overlap.
- This table is used when the binary is loaded. I.e.: Where does memory for .bss section with uninitialized global variables have to be allocated?

ELF



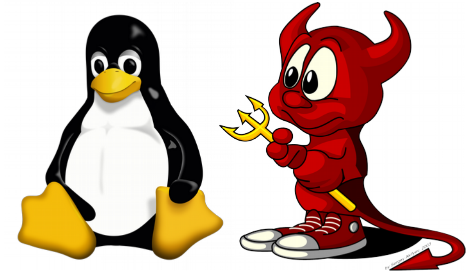
```
typedef struct
{
  Elf64_Word   sh_name;    /* Section name (string tbl index) */
  Elf64_Word   sh_type;    /* Section type */
  Elf64_Xword  sh_flags;      /* Section flags */
  Elf64_Addr   sh_addr;    /* Section virtual addr at execution */
  Elf64_Off sh_offset;     /* Section file offset */
  Elf64_Xword  sh_size;    /* Section size in bytes */
  Elf64_Word   sh_link;    /* Link to another section */
  Elf64_Word   sh_info;    /* Additional section information */
  Elf64_Xword  sh_addralign;     /* Section alignment */
  Elf64_Xword  sh_entsize;    /* Entry size if section holds table */
} Elf64_Shdr;
```

ELF



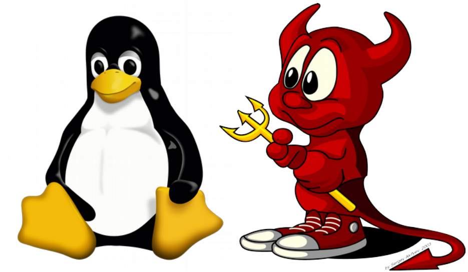
```
[martin@vmlinwork 1]$ readelf -S main
There are 35 section headers, starting at offset 0x2270:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400318  00000318
       000000000000003d  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400356  00000356
       0000000000000008  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400360  00000360
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400380  00000380
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003b0  000003b0
       0000000000000018  0000000000000018  AI       5    23     8
  [11] .init             PROGBITS         00000000004003c8  000003c8
       0000000000000017  0000000000000000  AX       0     0     4
```

ELF



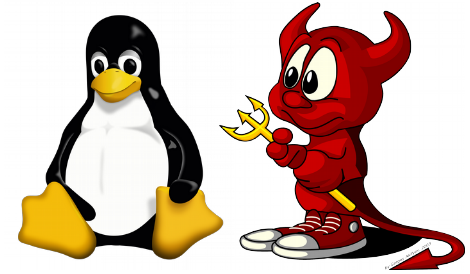
- Symbols table and Strings table
  - Symbol name (index in strings table)
  - Symbol type (not specified, function, object, file, section, TLS data, etc.)
  - Symbol visibility (local, global, weak, etc.)
  - Symbol section
  - Symbol value (I.e. offset where symbol is located within the section)
  - Symbol size

ELF



```
typedef struct
{
  Elf64_Word      st_name;      /* Symbol name (string tbl index) */
  unsigned char   st_info;      /* Symbol type and binding */
  unsigned char   st_other;     /* Symbol visibility */
  Elf64_Section   st_shndx;     /* Section index */
  Elf64_Addr      st_value;     /* Symbol value */
  Elf64_Xword     st_size;      /* Symbol size */
} Elf64_Sym;
```

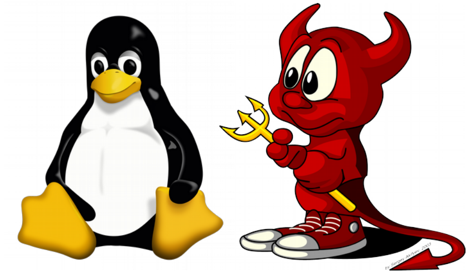
ELF



```
extern int var_a;

int var_a = 2;

static int var_b = 1;

extern int func_a (void);

static void func_b (void);

int main (void) {
    return func_a();
}

static void func_b (void) {
}
```

main.c

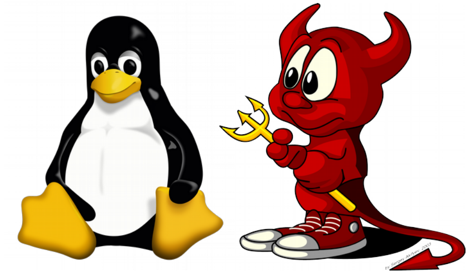
ELF



```
Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 
     5: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 var_b
     6: 000000000000000b     7 FUNC    LOCAL  DEFAULT    1 func_b
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6 
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7 
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 
    10: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 var_a
    11: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND func_a
```

main.o → gcc -o main.o main.c

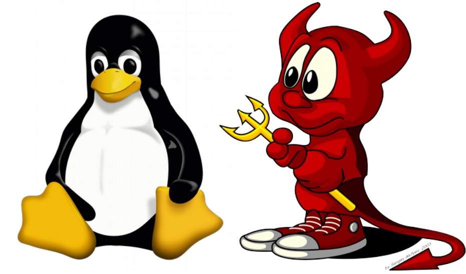
ELF



```
0000:0210 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0000:0220 00 6D 61 69  6E 2E 63 00  76 61 72 5F  62 00 66 75  .main.c.var b.fu
0000:0230 6E 63 5F 62  00 76 61 72  5F 61 00 6D  61 69 6E 00  nc b.var a.main.
0000:0240 66 75 6E 63  5F 61 00 00  05 00 00 00  00 00 00 00  func a..........
0000:0250 02 00 00 00  0C 00 00 00  FC FF FF FF  FF FF FF FF  ........üÿÿÿÿÿÿÿ
0000:0260 20 00 00 00  00 00 00 00  02 00 00 00  02 00 00 00   ...............
0000:0270 00 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00  ........@.......
0000:0280 02 00 00 00  02 00 00 00  0B 00 00 00  00 00 00 00  ................
```

main.o → string table

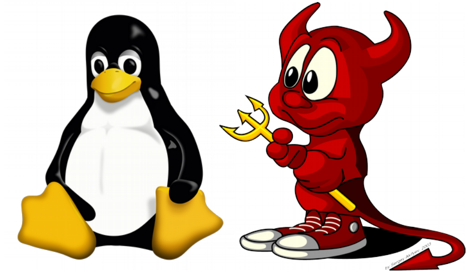
ELF



```
Symbol table '.symtab' contains 71 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND _
    57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    58: 0000000000601020     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    59: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    60: 00000000004005a8     0 OBJECT  GLOBAL HIDDEN    15 __dso_handle
    61: 00000000004005a0     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    62: 0000000000400520   101 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    63: 0000000000601028     0 NOTYPE  GLOBAL DEFAULT   25 _end
    64: 0000000000400400    43 FUNC    GLOBAL DEFAULT   13 _start
    65: 0000000000601024     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    66: 00000000004004f6    32 FUNC    GLOBAL DEFAULT   13 main
    67: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    68: 0000000000601028     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
```

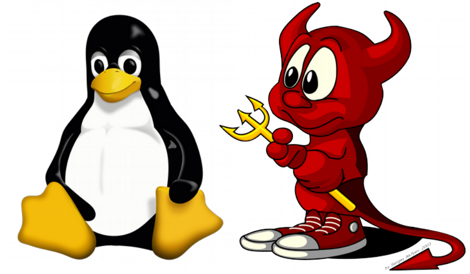
main → linked binary (against glibc)

ELF



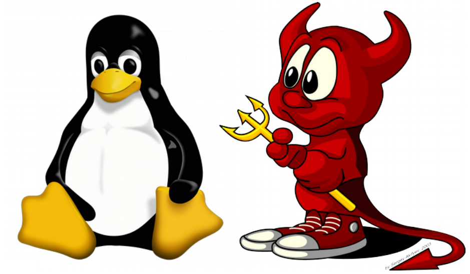
- Dynamic table
  - It’s a “map” to used by the dynamic linker in run time.
  - Required shared libraries (names)
  - Tables addresses in memory and sizes
    - Hashes, Strings, Symbols, Relocation, etc.
  - GOT and PLT tables addresses in memory
  - Initialization and finalization function addresses
  - Ends in NULL

ELF



- Is information redundant with Sections Table?
  - Sections Table is no mapped to the process memory.
  - Dynamic Table is mapped. Why? I.e.:
    - Kernel loads an executable binary
    - Control is transferred to the interpreter (dynamic loader)
    - Dynamic loader needs to know, among other things, which libraries does the executable binary link and map them to the process memory (example: libc)

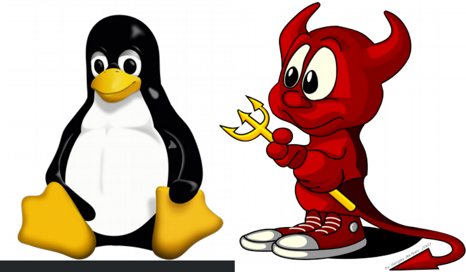
ELF



```
typedef struct
{
  Elf64_Sxword  d_tag;          /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;    /* Integer value */
      Elf64_Addr d_ptr;          /* Address value */
    } d_un;
} Elf64_Dyn;
```

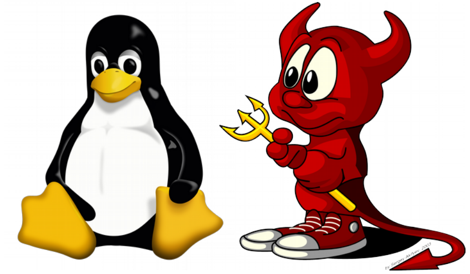
d_tag value may be: `DT_NULL, DT_NEEDED, DT_PLTGOT, DT_STRTAB, DT_SYMTAB, DT_RELA, DT_RELASZ, etc.`

ELF



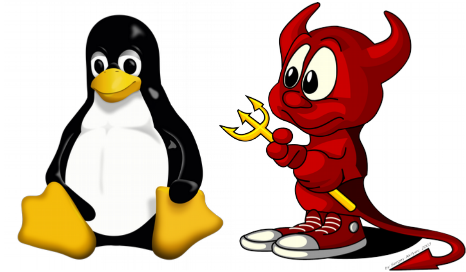
```
[martin@vmlinwork 1]$ readelf -d main

Dynamic section at offset 0xe20 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4003c8
 0x000000000000000d (FINI)               0x400594
 0x0000000000000019 (INIT_ARRAY)         0x600e08
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x600e10
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x400318
 0x0000000000000006 (SYMTAB)             0x4002b8
 0x000000000000000a (STRSZ)              61 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x601000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4003b0
 0x0000000000000007 (RELA)               0x400380
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400360
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x400356
 0x0000000000000000 (NULL)               0x0
```

ELF



If a dynamic loader has to load an executable binary and only knows its base virtual address, **how is it possible to reach the Dynamic Table?**

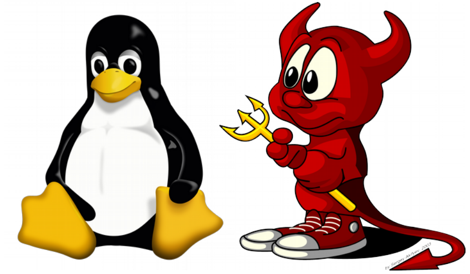
ELF



If a dynamic loader has to load an executable binary and only knows its base virtual address, **how is it possible to reach the Dynamic Table?**

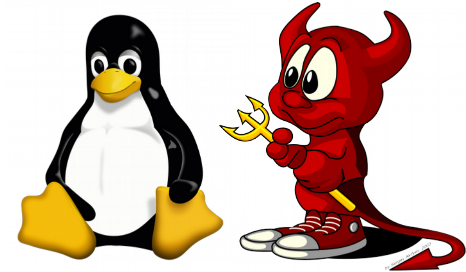
- Starts reading from the base address
- Jumps File Header (64 bytes) to get the Program Headers table
- Parses it and finds the Dynamic Table address
- Then it's possible, for example, to see which libraries are required (DT_NEEDED), locate library names (Symbols and Strings tables) and load them
- Kernel already did the heavy-lifting of mapping all these information in the executable binary file to virtual memory

ELF



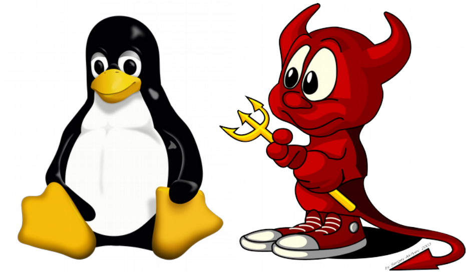
- Relocations Table
  - When an object calls an external function, the offset or address is not known until link time
  - Placeholder:
    - CALL ??? → + relocation entry
  - Then same happens for global variables present in external objects
  - When linking, placeholders are updated with the correct location

ELF



- Relocate a binary or a library
  - Libraries are usually PIC nowadays (Position Independent Code)
  - Binaries are usually not PIE in x86 and PIE in x86_64
  - If not PIC/PIE, there may be absolute addresses (i.e. CALL absolute-address). If PIC/PIE, relative addressing is used but relocations for the GOT/PLT tables is needed (after run time symbols resolution)
  - Not every binary is relocatable: depends on relocation information availability

ELF

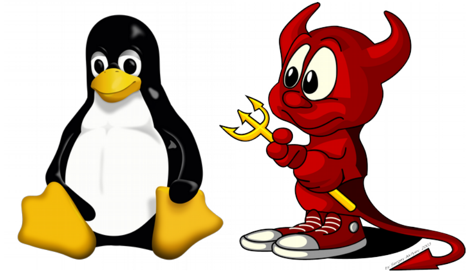


```
Relocation section '.rela.text' at offset 0x248 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name -
000000000005  000c00000002 R_X86_64_PC32     0000000000000000 func_a - 4

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   e8 00 00 00 00          callq  9 <main+0x9>
   9:   5d                      pop    %rbp
   a:   c3                      retq
```

main.o

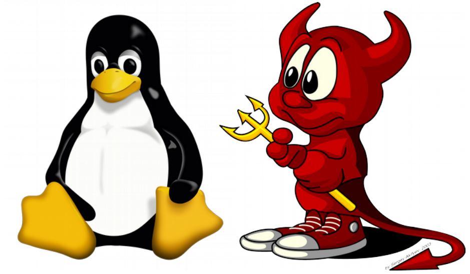
Relocation in .text: in offset 5 insert the value of the func_a symbol, after resolution (link time). Value is 32 bits long (call near).

ELF



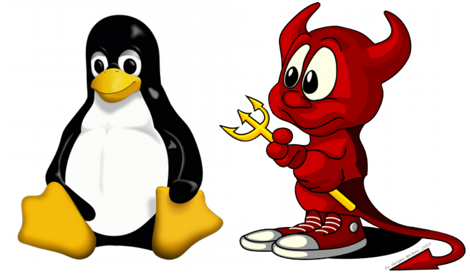
```
[martin@vmlinwork 1]$ readelf -r main

Relocation section '.rela.dyn' at offset 0x380 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000600ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x3b0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

ELF



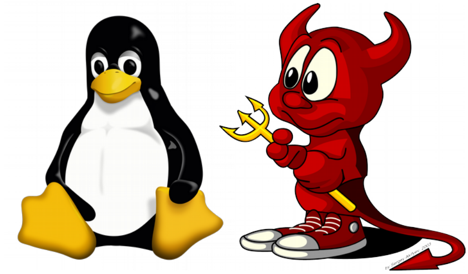
- Data section
  - .data
    - Initialized global variables
    - .rodata → read-only
  - .bss
    - Uninitialized global variables
    - Not in file, only in memory
    - Initialized with 0s

ELF



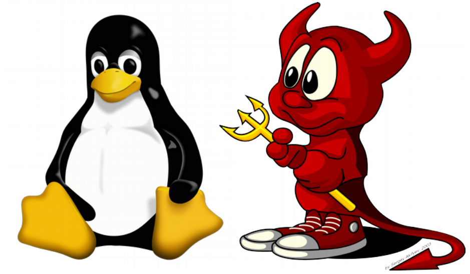
```
#include <stdio.h>

int a;
int b = 1;
const int c = 2;

int main(int argc, char* argv[]){
    printf("hola\n");
    return 0;
}
```

ELF



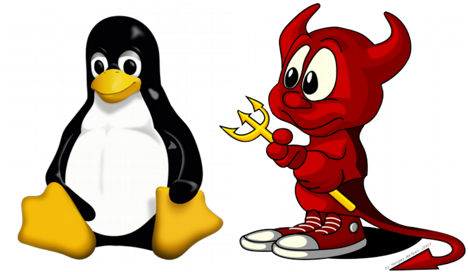
```
[martin@vmlinwork 1]$ readelf -x 25 main

Section '.bss' has no data to dump.
[martin@vmlinwork 1]$ readelf -x 24 main

Hex dump of section '.data':
  0x00601020 00000000 01000000                   ........

[martin@vmlinwork 1]$ readelf -x 15 main

Hex dump of section '.rodata':
  0x004005a0 01000200 00000000 00000000 00000000 ................
  0x004005b0 02000000 686f6c61 00                ....hola.
```

ELF



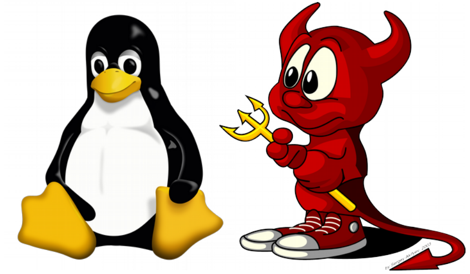
- Executable instructions: .text
  - Architecture opcodes
    - Functions implementation
    - *glibc* static stubs (if linked)
      - I.e. initialization routines like _*start* (<glibc>/sysdeps/x86_64/start.S) and __*libc_csu_init* (<glibc>/csu/elf-init.c)
- Readable and executable segment (not writable)

ELF



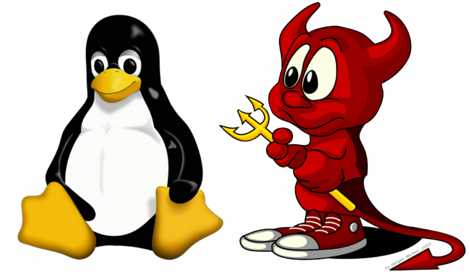
```
[martin@vmlinwork 1]$ objdump -d main | grep -A 14 -e "<_start>"
0000000000400400 <_start>:
  400400:       31 ed                   xor    %ebp,%ebp
  400402:       49 89 d1                mov    %rdx,%r9
  400405:       5e                      pop    %rsi
  400406:       48 89 e2                mov    %rsp,%rdx
  400409:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40040d:       50                      push   %rax
  40040e:       54                      push   %rsp
  40040f:       49 c7 c0 90 05 40 00    mov    $0x400590,%r8
  400416:       48 c7 c1 20 05 40 00    mov    $0x400520,%rcx
  40041d:       48 c7 c7 f6 04 40 00    mov    $0x4004f6,%rdi
  400424:       ff 15 c6 0b 20 00       callq  *0x200bc6(%rip)        # 600ff0 <_DYNAMIC+0x1d0>
  40042a:       f4                      hlt
  40042b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

ELF



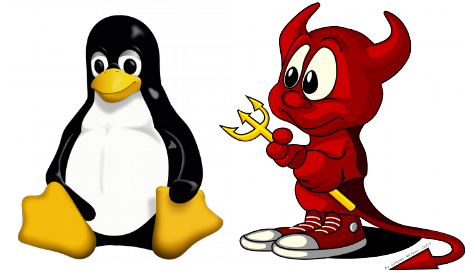
- Global Offset Table (GOT)
  - A binary or library may use global variables and functions exported by other libraries
  - Libraries are Position-Independent-Code (PIC) nowadays
  - When a binary or library are compiled, the virtual address of external functions and global variables in not known
  - CALL ??? → how can we fix this?

ELF



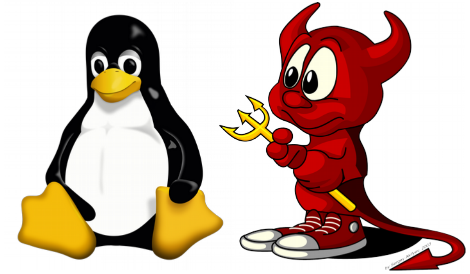
- Global Offset Table (GOT)
  - Table in memory for each binary and library
  - Memory address of extern functions and variables (from other libraries)
  - It’s filled in *runtime* with absolute values (virtual addresses)
  - MOV *<variable-GOT-entry>
    - Addressing relative to GOT

ELF



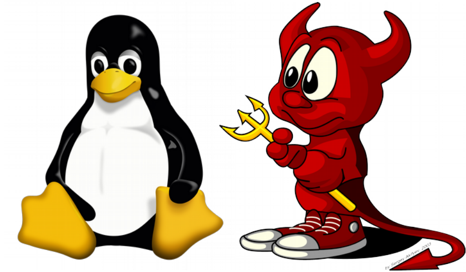
- Procedure Linkage Table (PLT)
  - Trampolines to call external functions using the GOT (one trampoline per external function)
  - Executable code (executable segment)
  - Instead of CALL *<function-GOT-entry>, it is CALL/JMP <function-PLT-entry>

ELF



- Procedure Linkage Table (PLT)
  - Trampoline executes JMP *<function-GOT-entry>
  - When GOT entry is resolved, jumps to the function
  - When GOT entry is not resolved (initial state), trampoline invokes the dynamic loader for resolution

ELF



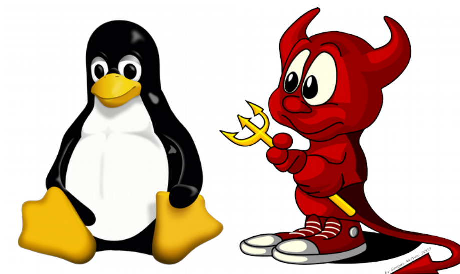
```
int main(void) {
    int ret = -1;
    pthread_t thread_info;
    pthread_attr_t thread_attr;

    printf("Main thread PID: %d\n", getpid());
    printf("Main thread TID: %d\n", syscall(__NR_gettid));
```

```
0000000000400a59:   push    %rbp
0000000000400a5a:   mov     %rsp,%rbp
0000000000400a5d:   sub     $0x60,%rsp
 51                   int ret = -1;
0000000000400a61:   movl    $0xffffffff,-0x4(%rbp)
 55                   printf("Main thread PID: %d\n", getpid
0000000000400a68:   callq   0x4007e0 <getpid@plt>
```

“main” ELF (x86_64) - getpid call example through PLT

PLT segment

ELF



```
00000000004007ac:   nopl    0x0(%rax)
                  pthread_create@plt:
00000000004007b0:   jmpq    *0x201862(%rip)        # 0x602018
00000000004007b6:   pushq   $0x0
00000000004007bb:   jmpq    0x4007a0
                  puts@plt:
00000000004007c0:   jmpq    *0x20185a(%rip)        # 0x602020
00000000004007c6:   pushq   $0x1
00000000004007cb:   jmpq    0x4007a0
                  sigaction@plt:
00000000004007d0:   jmpq    *0x201852(%rip)        # 0x602028
00000000004007d6:   pushq   $0x2
00000000004007db:   jmpq    0x4007a0
                  getpid@plt:
00000000004007e0:   jmpq    *0x20184a(%rip)        # 0x602030
00000000004007e6:   pushq   $0x3
00000000004007eb:   jmpq    0x4007a0
                  printf@plt:
```

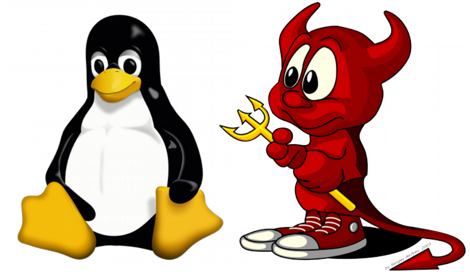
Relocation entry number

go to dynamic loader

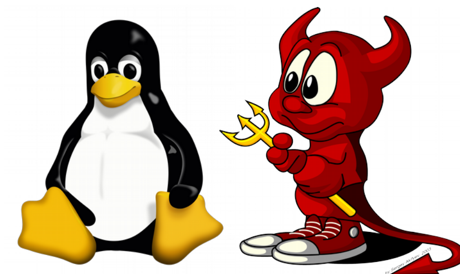
Take the address from GOT PLT entry

```
Relocation section '.rela.plt' at offset 0x668 contains 12 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000602018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 pthread_create@GLIBC_2.2.5 +
000000602020  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000602028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 sigaction@GLIBC_2.2.5 + 0
000000602030  000500000007 R_X86_64_JUMP_SLO 0000000000000000 getpid@GLIBC_2.2.5 + 0
000000602038  000600000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000602040  000800000007 R_X86_64_JUMP_SLO 0000000000000000 pthread_attr_init@GLIBC_2.2.5
000000602048  000900000007 R_X86_64_JUMP_SLO 0000000000000000 syscall@GLIBC_2.2.5 + 0
000000602050  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 sigemptyset@GLIBC_2.2.5 + 0
```

Dynamic Loader

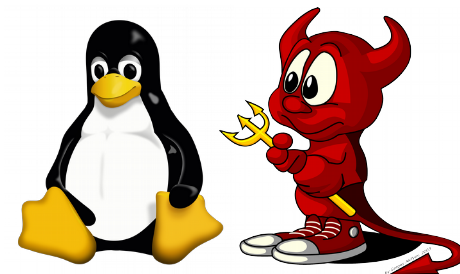


- Dynamic loader
  - As an interpreter, executes first
    - Then control is transferred to the binary entry point
  - Loads dynamically linked libraries
  - Resolves symbols
  - Loads libraries in runtime (dlopen)
  - Fills the GOT table in runtime (uses relocation information to determine where to write memory addresses once symbols are resolved)



Demo 1.1

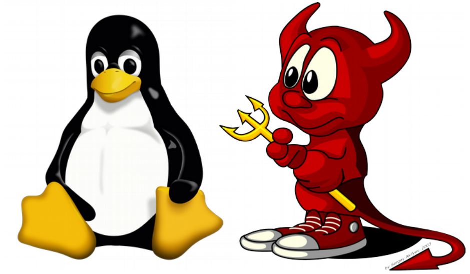
Kernel parsing ELF format to launch an executable binary (sys_execve)



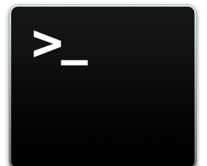
Demo 1.2

GOT + PLT call (dynamic linker)

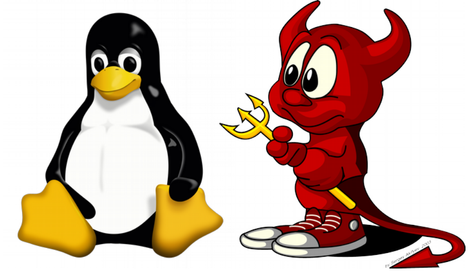
ELF



- $ gcc -o main -g main.c
- $ echo $?
- $ readelf -a ./main
- $ objdump -d ./main | grep -A 500 -e "<_start>"
- $ gdb main
  - (gdb) break main
  - (gdb) run
  - (gdb) x/10xb 0x400000
- $ cat /proc/<PID>/maps

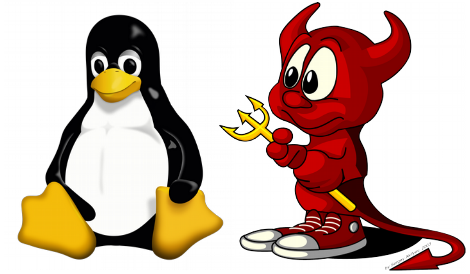


Lab 1.1



- Patch “main” to:
  - return 5 (instead of 0)
  - infinitely loop
  - print “ʊ” to *stdout*
- Debug with *gdb* each case
- Patch with gdb in runtime each of the previous cases

Lab 1.2



- Debug the dynamic loader parsing ELF format when loading a shared library
- Debug the kernel parsing ELF format when a module (.ko) is loaded

References

- https://refspecs.linuxfoundation.org
- https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=elf/elf.h;hb=HEAD
- https://linux.die.net/man/5/elf