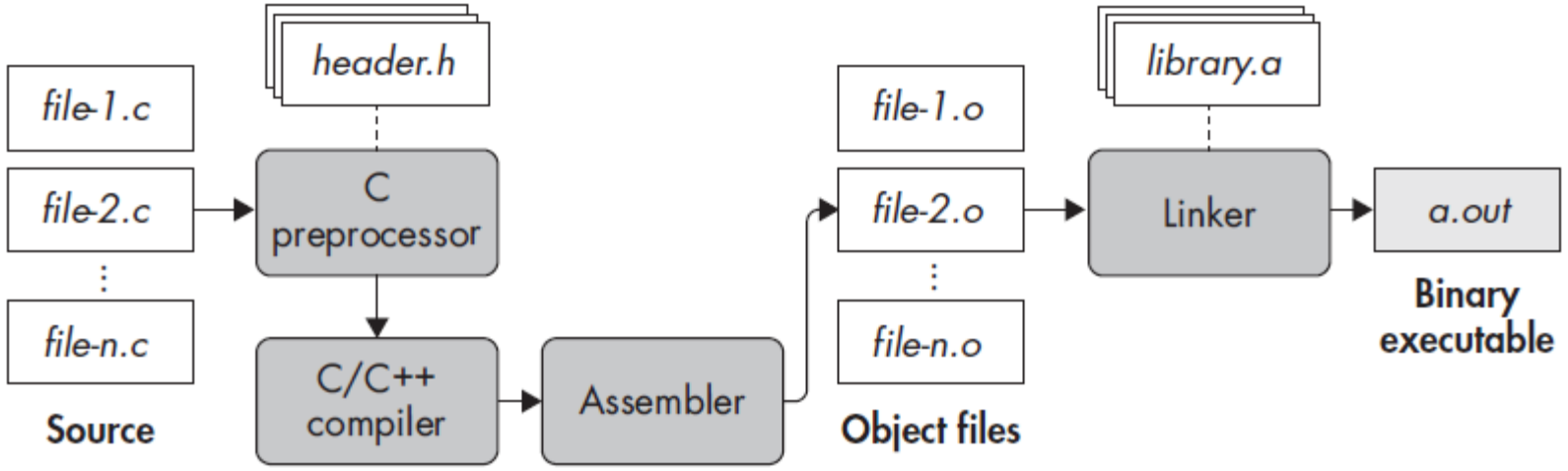
# CS 7/820
# Binary Anatomy

# The C Compilation Process

- Binaries are produced through compilation which is the process of translating human-readable code like C/C++ into machine code that the processor can execute
- Compiling C code involves four phases, preprocessing, compilation, assembly, linking



# Preprocessing phase

- Expands #define and #include directives int source file to leave behind pure source code to be compiled
- To see the output of the pre-processor: gcc –E – P compilation_example.c
  - -E to stop after pre-processing
  - -F to omit debugging information

```
#define FORMAT_STRING "%s"
#define OUTPUT "Hello, world\n"

#include <stdio.h>

int
main(int argc, char *argv[])
{
        printf(FORMAT_STRING, OUTPUT);
        return 0;
}
```

```
typedef long unsigned int size_t;
typedef __builtin_va_list __gnuc_va_list;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
.....................
.....................
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
extern int remove (const char *__filename) __attribute__ ((__nothrow__ , __leaf__));
extern int rename (const char *__old, const char *__new) __attribute__ ((__nothrow__ , __leaf__));
extern int renameat (int __oldfd, const char *__old, int __newfd,
       const char *__new) __attribute__ ((__nothrow__ , __leaf__));
extern int fgetc_unlocked (FILE *__stream);
extern int fputc (int __c, FILE *__stream);
extern int putc (int __c, FILE *__stream);
extern int putchar (int __c);
extern int feof_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int ferror_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern void perror (const char *__s);
extern int sys_nerr;
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
.....................
.....................
int
main(int argc, char *argv[])
{
 printf("%s", "Hello, world\n");
 return 0;
}
```

# Compilation phase

- Takes preprocessed code and translates into assembly language
  - Optimization occurs typically configurable (-O0 –O3), may have a profound effect on disassembly
- Advantage of translating into assembly language is to allow one assembler to be used for multiple languages
- Use gcc –S to generate the assembly file and –masm=intel to generate the assembly in Intel syntax (not AT&T)
- Constants and variables have “symbolic names” rather than just addresses whether it’s nameless or has an explicit label

# Compilation phase

- Note .LC0, main, puts

```
        .file   "main.c"
        .intel_syntax noprefix
        .text
        .section        .rodata
.LC0:
        .string "Hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     QWORD PTR [rbp-16], rsi
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 9.2.1 20190827 (Red Hat 9.2.1-1)"
        .section        .note.GNU-stack,"",@progbits
```

# Assembly phase

- Generate some real machine code!
- Input is a set of assembly language files, output is a set of ***object*** files
- Object files have machine code but cannot be executed by the processor just yet (need an executable)
- gcc –c main.c

  -c creates and retains the object file with the .o extension
- Use the ***file*** utility to see what kind of file it might be:

  file main.o

```
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

# Assembly phase

```
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

- ELF
  - Conforms to the ELF specification for binary executables
- It’s a 64-bit ELF file (compiling for x86-64)
- LSB
  - Ordering in memory with least significant byte first
- File is relocatable
  - Relocatable files don’t rely on being placed at any particular address in memory
  - They can be moved around at will
  - All *binary* object files are typically relocatable (unlike a *binary* executable)
  - Fact that object files are compiled independent of each other makes them relocatable by default

# Linking phase

- Final phase of the compilation process, linker links all object files into a single binary executable
- Generally incorporates a (link-time) optimization process
- With object files referencing other object files or libraries, no object will end up with a particular base address (relocatable feature)
- Object files only contain ‘relocation symbols’ that specify function or variable references (called symbolic references) that are external to the program, which would be eventually resolved
- Linker’s job is to take all the object files belonging to a program and merge them into a single coherent executable (typically intended to be loaded into a particular memory address)
- Linker resolves most symbolic references, library references may or may not be resolved (depends on the type of library)

# Linking phase

- Static libraries (on Linux they typically have the .a extension) are merged into the binary executable
- Dynamic (shared) libraries are shared in memory among programs, they are loaded only once into memory
- Any binary that wants to use a dynamic library has to use this shared copy
- Addresses of dynamic libraries are not known during linking so references to them are not resolved until the binary is actually loaded into memory to be executed
- Use the ***file*** utility against the executable (default name is a.out which can be changed with the –o switch to gcc)

# Linking phase

file a.out

```
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=51f9374888b6fee40949f02b072a34556a5e501e, for GNU
/Linux 3.2.0, not stripped
```

- ELF format as before but it's an executable (not relocatable), 64-bit LSB
  - Conforms to the ELF specification for binary executables
- Dynamically linked
  - Some libraries are not merged into the executable, they are shared
- Interpreter /lib64/ld-linux-x86-64.so.2 is the dynamic linker that will be used to resolve the final dependencies when it is executed
- What does 'not stripped' mean?

# Symbols

- Compilers generate symbols to keep track of symbolic names and record which binary code and data correspond to each symbol
- Example: Function symbols provide mapping from symbolic, high-level function names to the first address and the size of each function
  - Useful when linker needs to combine object files and resolve references
- ***readelf*** is a utility to view binary files (-s or -- syms flag dumps out the symbol table)

  readelf -s a.out
- Linkers need only basic symbols but debugging symbols are also possible to be generated (ELF binaries typically generate these in DWARF format)

# Symbols

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 85 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000004002a8     0 SECTION LOCAL  DEFAULT    1
     2: 00000000004002c4     0 SECTION LOCAL  DEFAULT    2
     3: 00000000004002e8     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400308     0 SECTION LOCAL  DEFAULT    4
    ....................................................
    ....................................................
    66: 0000000000404000     0 OBJECT  LOCAL  DEFAULT   22 _GLOBAL_OFFSET_TABLE_
    67: 00000000004011c0     5 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    68: 0000000000404020     0 NOTYPE  WEAK   DEFAULT   23 data_start
    69: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
    70: 0000000000404024     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    71: 00000000004011c8     0 FUNC    GLOBAL HIDDEN    14 _fini
    72: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    73: 0000000000404020     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    74: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    75: 0000000000402008     0 OBJECT  GLOBAL HIDDEN    15 __dso_handle
    76: 0000000000402000     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    77: 0000000000401150   101 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    78: 0000000000404028     0 NOTYPE  GLOBAL DEFAULT   24 _end
    79: 0000000000401070     5 FUNC    GLOBAL HIDDEN    13 _dl_relocate_static_pie
    80: 0000000000401040    47 FUNC    GLOBAL DEFAULT   13 _start
    81: 0000000000404024     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    82: 0000000000401126    32 FUNC    GLOBAL DEFAULT   13 main
    83: 0000000000404028     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
    84: 0000000000401000     0 FUNC    GLOBAL HIDDEN    11 _init
```

# Symbolic Information

- Very useful for binary analysis
- Disassembly becomes much easier having a set of well-defined function symbols
- Prevents disassembly data as code
- Makes it easier for reverse engineering with even basic symbols (not necessarily debugging symbols)
- In production-ready binaries, symbols are ***stripped***
  - Debugging information should definitely be left out
  - Basic symbols also left out often to reduce file size and also to prevent reverse engineering

# Stripped binary

- Default behavior of gcc is to not automatically strip
- ***strip*** utility ---> strip a.out
  - Useful for minimizing their file size, streamlining them for distribution
  - Make it more difficult to reverse-engineer the compiled code
  - Only a few symbols that are used to resolve dynamic dependencies are left

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

# Disassembling a Binary (object file)

- ***objdump*** utility will show you disassembled output
  - objdump –sj  .rodata main.o ---> show only contents of .rodata section

```
main.o:     file format elf64-x86-64

Contents of section .rodata:
 0000 48656c6c 6f2c2077 6f726c64 00        Hello, world.
```

  - objdump –M intel -d  .rodata main.o ---> show contents of executable sections in Intel format

```
main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   48 83 ec 10             sub    rsp,0x10
   8:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   b:   48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
   f:   bf 00 00 00 00          mov    edi,0x0
  14:   e8 00 00 00 00          call   19 <main+0x19>
  19:   b8 00 00 00 00          mov    eax,0x0
  1e:   c9                      leave
  1f:   c3                      ret
```

# Disassembling a Binary (object file)

- Assembly File Vs Disassembled Output

```
.cfi_startproc
push    rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
mov     rbp, rsp
.cfi_def_cfa_register 6
sub     rsp, 16
mov     DWORD PTR [rbp-4], edi
mov     QWORD PTR [rbp-16], rsi
mov     edi, OFFSET FLAT:.LC0
call    puts
mov     eax, 0
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

```
0000000000000000 <main>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   48 83 ec 10             sub    rsp,0x10
   8:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   b:   48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
   f:   bf 00 00 00 00          mov    edi,0x0
  14:   e8 00 00 00 00          call   19 <main+0x19>
  19:   b8 00 00 00 00          mov    eax,0x0
  1e:   c9                      leave
  1f:   c3                      ret
```

- Use readelf on the object file to check relocatable symbol information
- readelf --relocs main.o

```
Relocation section '.rela.text' at offset 0x1f8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000010  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
000000000015  000a00000004 R_X86_64_PLT32    0000000000000000 puts - 4
```

# Disassembling a Binary (executable)

- Will contain a lot more functions and code sections than the object file
  - Sections for program initialization and calling shared libraries
  - .text section is the main code section and contains the main function, as well as other functions like _start for setting up command line arguments
- Previously unresolved references have now been resolved
- Binary executable contains significantly more code and data
- objdump –M intel –d a.out

```
Disassembly of section .init:

0000000000401000 <.init>:
  401000:       f3 0f 1e fa             endbr64
  401004:       48 83 ec 08             sub    rsp,0x8
  401008:       48 8b 05 e9 2f 00 00    mov    rax,QWORD PTR [rip+0x2fe9]        # 403ff8 <puts@plt+0x2fc8>
  40100f:       48 85 c0                test   rax,rax
  401012:       74 02                   je     401016 <puts@plt-0x1a>
  401014:       ff d0                   call   rax
  401016:       48 83 c4 08             add    rsp,0x8
  40101a:       c3                      ret

Disassembly of section .plt:

0000000000401020 <puts@plt-0x10>:
  401020:       ff 35 e2 2f 00 00       push   QWORD PTR [rip+0x2fe2]        # 404008 <puts@plt+0x2fd8>
  401026:       ff 25 e4 2f 00 00       jmp    QWORD PTR [rip+0x2fe4]        # 404010 <puts@plt+0x2fe0>
  40102c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000401030 <puts@plt>:
  401030:       ff 25 e2 2f 00 00       jmp    QWORD PTR [rip+0x2fe2]        # 404018 <puts@plt+0x2fe8>
  401036:       68 00 00 00 00          push   0x0
  40103b:       e9 e0 ff ff ff          jmp    401020 <puts@plt-0x10>
```

# Disassembling a Binary (executable)

- Will contain a lot more functions and code sections than the object file
  - Sections for program initialization and calling shared libraries
  - .text section is the main code section and contains the main function, as well as other functions like _start for setting up command line arguments
- Previously unresolved references have now been resolved
- Binary executable contains significantly more code and data
- objdump –M intel –d a.out

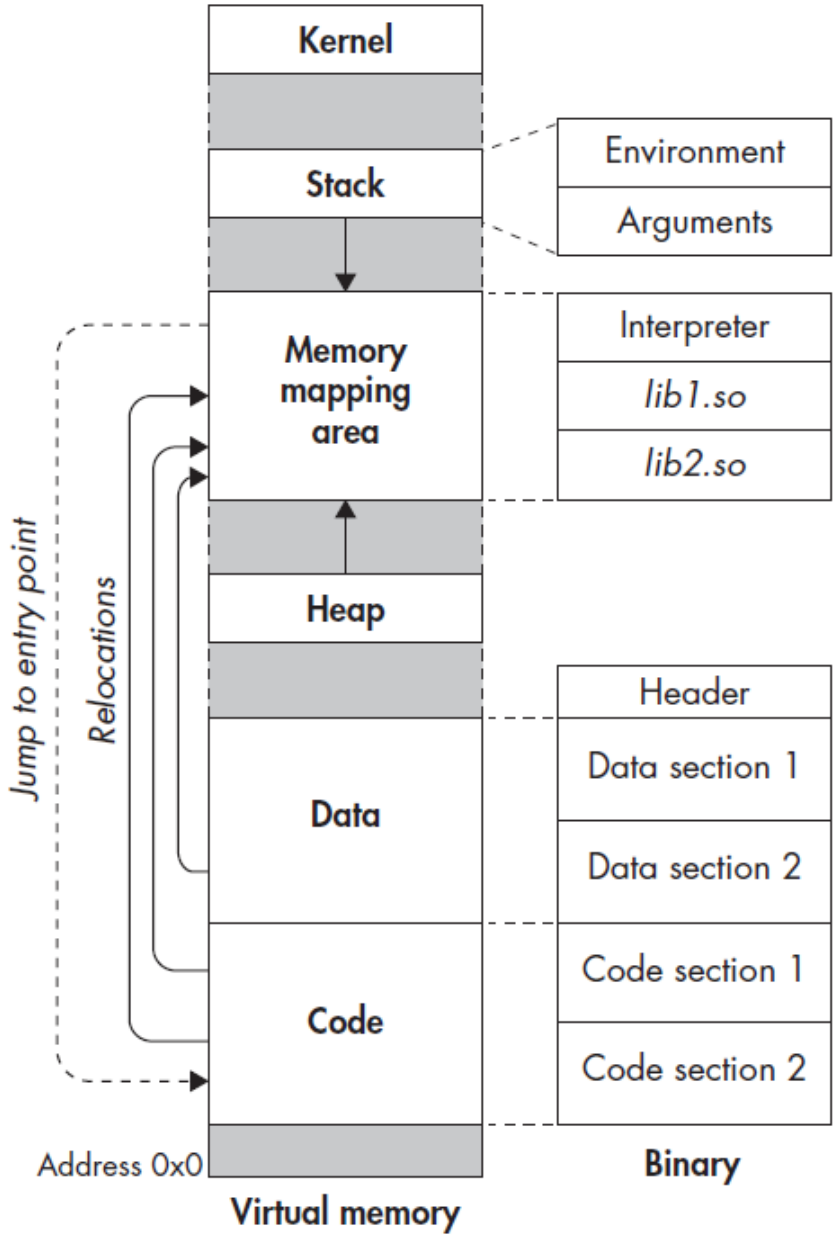
```
Disassembly of section .init:

0000000000401000 <.init>:
  401000:       f3 0f 1e fa             endbr64
  401004:       48 83 ec 08             sub    rsp,0x8
  401008:       48 8b 05 e9 2f 00 00    mov    rax,QWORD PTR [rip+0x2fe9]        # 403ff8 <puts@plt+0x2fc8>
  40100f:       48 85 c0                test   rax,rax
  401012:       74 02                   je     401016 <puts@plt-0x1a>
  401014:       ff d0                   call   rax
  401016:       48 83 c4 08             add    rsp,0x8
  40101a:       c3                      ret

Disassembly of section .plt:

0000000000401020 <puts@plt-0x10>:
  401020:       ff 35 e2 2f 00 00       push   QWORD PTR [rip+0x2fe2]        # 404008 <puts@plt+0x2fd8>
  401026:       ff 25 e4 2f 00 00       jmp    QWORD PTR [rip+0x2fe4]        # 404010 <puts@plt+0x2fe0>
  40102c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000401030 <puts@plt>:
  401030:       ff 25 e2 2f 00 00       jmp    QWORD PTR [rip+0x2fe2]        # 404018 <puts@plt+0x2fe8>
  401036:       68 00 00 00 00          push   0x0
  40103b:       e9 e0 ff ff ff          jmp    401020 <puts@plt-0x10>
```

# Disassembling a Binary (executable)

```
Disassembly of section .text:

0000000000401040 <_start>:
  401040:       f3 0f 1e fa             endbr64
  401044:       31 ed                   xor    ebp,ebp
  401046:       49 89 d1                mov    r9,rdx
  401049:       5e                      pop    rsi
  40104a:       48 89 e2                mov    rdx,rsp
  40104d:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
  401051:       50                      push   rax
  401052:       54                      push   rsp
  401053:       49 c7 c0 c0 11 40 00    mov    r8,0x4011c0
  40105a:       48 c7 c1 50 11 40 00    mov    rcx,0x401150
  401061:       48 c7 c7 26 11 40 00    mov    rdi,0x401126
  401068:       ff 15 82 2f 00 00       call   QWORD PTR [rip+0x2f82]        # 4
03ff0 <__libc_start_main@GLIBC_2.2.5>
  40106e:       f4                      hlt
..................
..................

0000000000401120 <frame_dummy>:
  401120:       f3 0f 1e fa             endbr64
  401124:       eb 8a                   jmp    4010b0 <register_tm_clones>

0000000000401126 <main>:
  401126:       55                      push   rbp
  401127:       48 89 e5                mov    rbp,rsp
  40112a:       48 83 ec 10             sub    rsp,0x10
  40112e:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401131:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  401135:       bf 10 20 40 00          mov    edi,0x402010
  40113a:       e8 f1 fe ff ff          call   401030 <puts@plt>
  40113f:       b8 00 00 00 00          mov    eax,0x0
  401144:       c9                      leave
  401145:       c3                      ret
  401146:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  40114d:       00 00 00
..................
..................
00000000004011c0 <__libc_csu_fini>:
  4011c0:       f3 0f 1e fa             endbr64
  4011c4:       c3                      ret

Disassembly of section .fini:

00000000004011c8 <_fini>:
  4011c8:       f3 0f 1e fa             endbr64
  4011cc:       48 83 ec 08             sub    rsp,0x8
  4011d0:       48 83 c4 08             add    rsp,0x8
  4011d4:       c3                      ret
```

- *Disassembling a stripped binary will still give us the different sections but the functions are not distinguishable*
  - *All functions are combined into one single blob except for the functions that are still distinct in the .plt section for those coming from dynamically linked shared libraries*

# Loading and Executing a Binary




- Details vary with platform and binary format but process of loading and executing is very similar
- O.S sets up a new process including a virtual address space
- An interpreter is mapped which knows how to load and perform necessary relocations (On Linux it is ld-linux.so – the ELF binary comes with a section called .interp which indicates the path of this interpreter)
- Resolving references to dynamic libraries is deferred (lazy binding)
- Interpreter looks up entry point and transfers control to it, beginning normal execution of the binary