Lecture 4

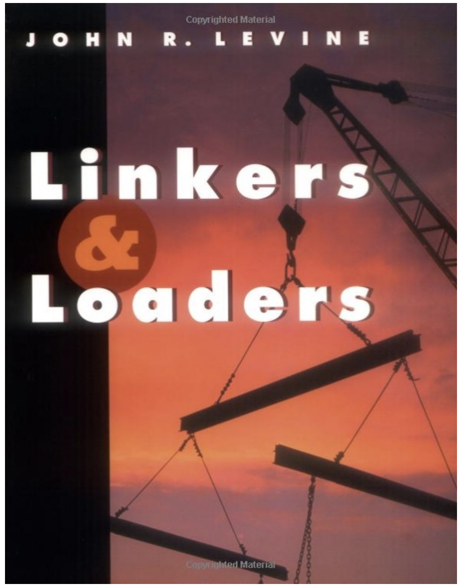
Linking and ELF

(A small digression)

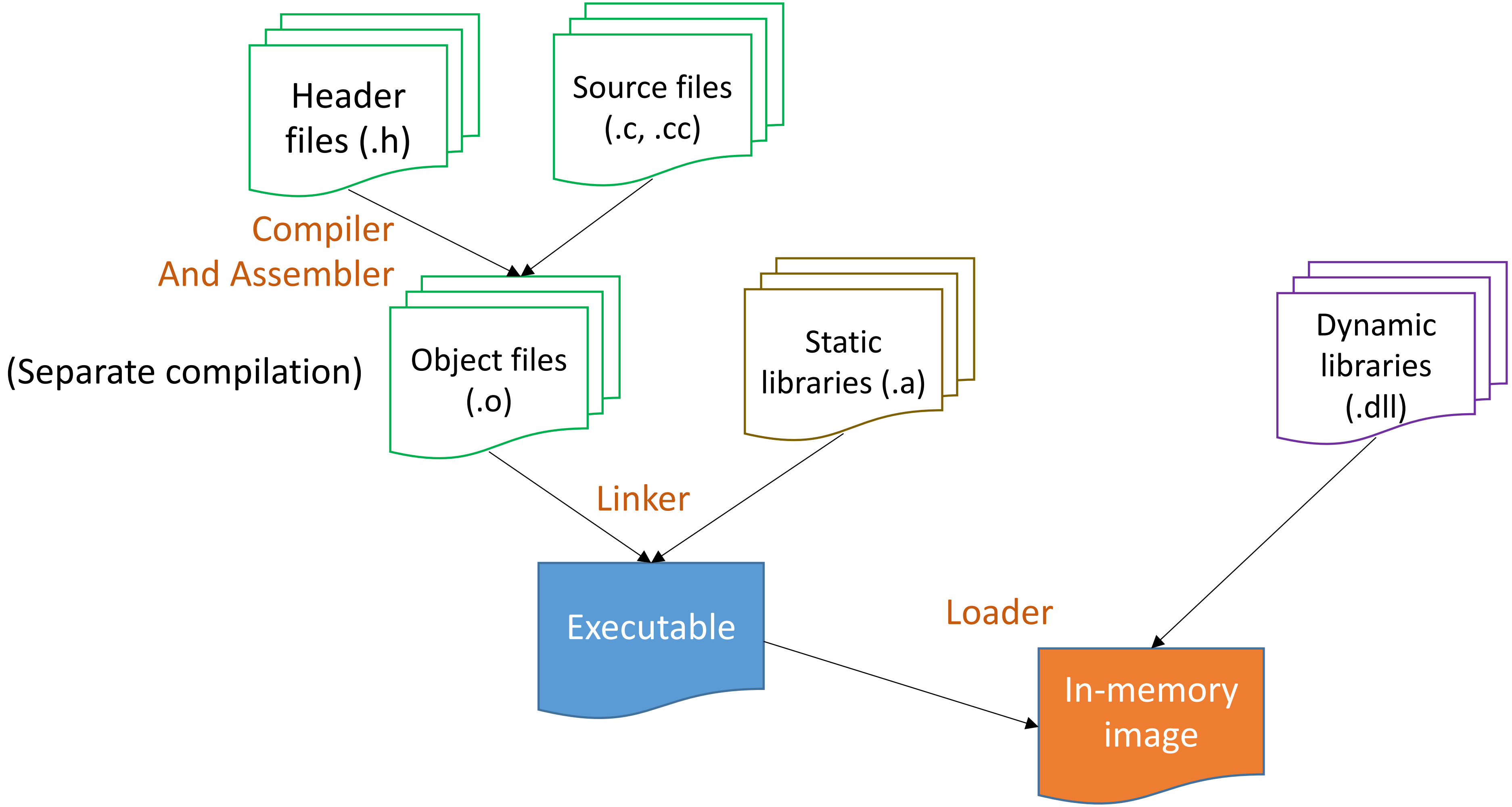
Learning Objectives

- Learn about the linking and loader process
  - What are they and why they are designed in that way

- Learn the Executable and Linking Format (ELF)



The compilation flow



Main functions

- High level programming languages are good for human productivity
  - But they are not what the machine understands
- Compilers: translate code written in high-level programming languages to assembly code of the target machine's ISA
- Assemblers: translate human readable assembly code to binary object files
- Linker: combines multiple object files and library modules into a single executable
- Loader: reads the executable and together with dynamic libraries, constructs a memory image of the application

Important design considerations

- Separate compilation
  - Allows for a better organization of the source code
    - Also facilitates sharing during code writing
  - Only need to recompile changed code
- External linkages
  - Modern applications are built on layers of libraries

Executable files

- Information needs to be conveyed to the OS
- The OS expects executable files to have a specific format
  - *Header info*
    - Code locations
    - Data locations
  - Code & data
  - *Symbol Table*
    - List of names of things defined in your program and where they are defined
    - List of names of things defined elsewhere that are used by your program, and where they are used.

Example – definition and use

```
#include <stdio.h>

int main () {

 printf (“hello, world\n”)

}
```

- Symbol defined in your program and used elsewhere
  - `main`
- Symbol defined elsewhere and used by your program
  - `printf`

A two-step operation

- Linking: Combining a set of programs, including library routines, to create a *loadable* image
  a) Resolving symbols defined within the set
  b) Listing symbols needing to be resolved by loader

- Loading: Starting from the loadable file, copy in dynamic libraries and construct the memory image in a new process

Key activities involved

- Relocation: assigning load addresses to various parts of a program and adjusting the code and data in the program to reflect such assignments
- Symbol resolution: maps symbols into the actual location in a memory image
- Loading: constructing a memory image in a OS process from the executable on secondary storage

Executable and Linking File Format

(ELF)

ELF

- Support for cross-compilation, dynamic linking, initializer/finalizer (e.g., the constructor and destructor in C++) and other advanced system features

- ELF has been adopted by FreeBSD and Linux as the current standard

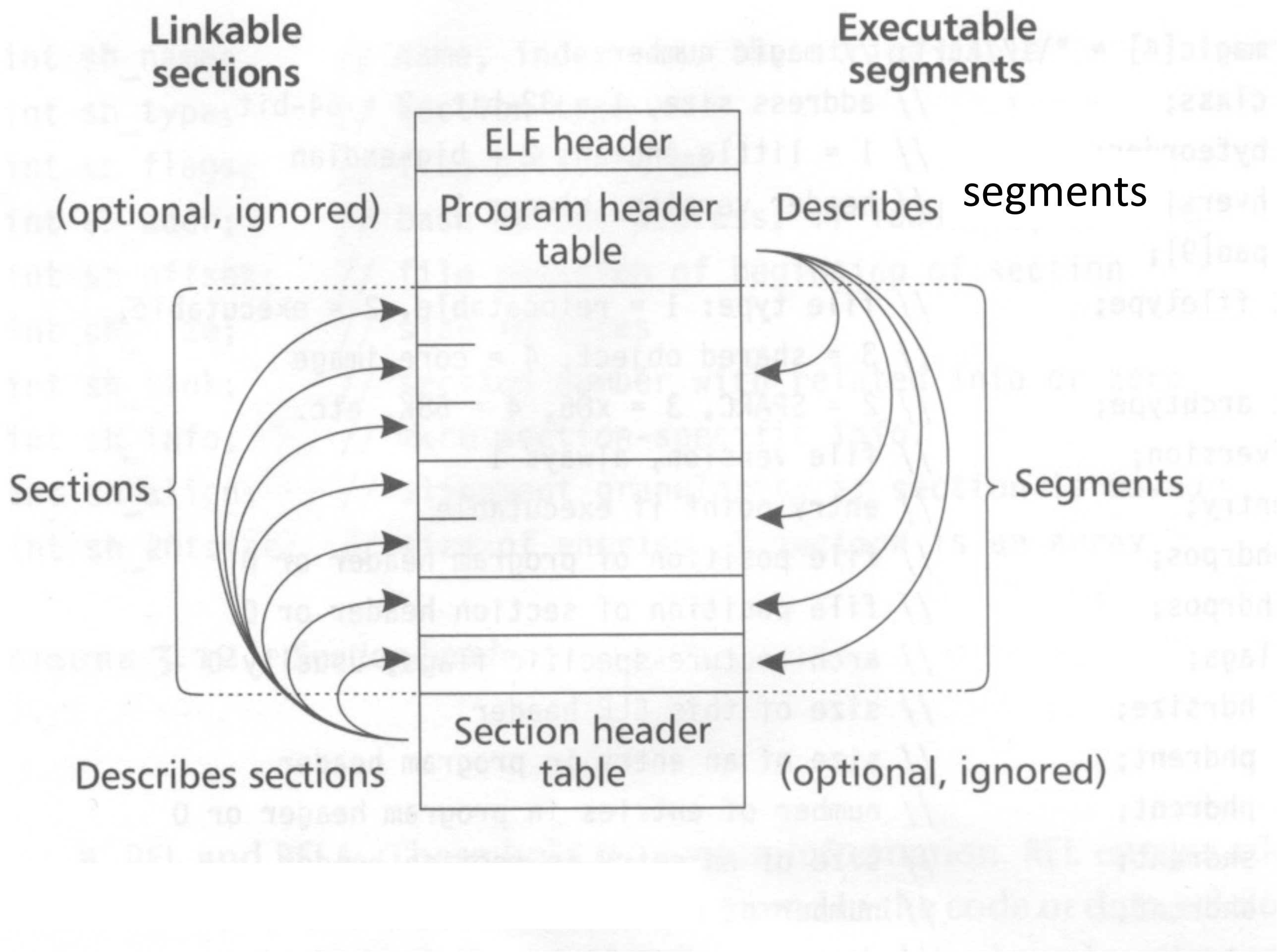
- ELF32 for 32-bit binaries and ELF64 for 64-bit binaries
  - Almost identical except for length of data

ELF File Types

- Relocatable (object files)
  - Created by compilers or assemblers. Need to be processed by the linker before running. Addresses assumed to start at zero.
- Executable
  - All relocation done and all symbol resolved except for shared library symbols that must be resolved at run time
- Shared object
  - Shared library containing both symbol information for the linker and directly runnable code
- Core file
  - For core dumps

ELF Structure

- Compilers, assemblers, and linkers treat the file as a set of logical sections described by a section header table.
- The system loader treats the file as a set of segments described by a program header table.


Linkable sections
Executable segments
ELF header
(optional, ignored)
Program header table
Describes
segments
Sections
Segments
Section header table
Describes sections
(optional, ignored)

ELF Structure

- A single segment usually consist of several sections. E.g., a loadable read-only segment could contain sections for executable code, read-only data, and symbols for the dynamic linker.
- Relocatable files have section header tables. Executable files have program header tables. Shared object files have both.
- Sections are intended for further processing by a linker, while the segments are intended to be mapped into memory.
- See **`/usr/include/elf.h`**

ELF Header

- The ELF header is always at offset zero of the file.
- The program header table and the section header table's offset in the file are defined in the ELF header.
  - Use `fseek()` to find.
- The header is decodable even on machines with a different byte order from the file's target architecture.
  - After reading class and byteorder fields, the rest fields in the ELF header can be decoded.
  - ELF supports two different address sizes:
    - 32 bits
    - 64 bits

ELF Program Header (32 bits)

```
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;
```

4-byte magic number is “0x7F” followed by the string “ELF”.

Relocatable Files

- A relocatable or shared object file is a collection of sections.
- Each section contains a single type of information, such as program code, read-only data,or read/write data, relocation entries, or symbols.
- Every symbol’s address is defined relative to a section.
  - Therefore, a procedure’s entry point is relative to the program code section that contains that procedure’s code.

Section Header

```
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;
```

Types in Section Header (`sh_type`)

- PROGBITS: This holds program contents including code, data, and debugger information.
- NOBITS: Like PROGBITS. However, it occupies no space.
- SYMTAB and DYNSYM: These hold symbol table.
- STRTAB: This is a string table.
- REL and RELA: These hold relocation information.
- DYNAMIC and HASH: This holds information related to dynamic linking.

Flags in Section Header (`sh_flags`)

- WRITE: This section contains data that is writable during process execution.
- ALLOC: This section occupies memory during process execution.
- EXECINSTR: This section contains executable machine instructions.

Sections

- `.text`:
  - This section holds executable instructions of a program.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- `.data`:
  - This section holds initialized data that contributes to the program's image.
  - Type: PROGBITS
  - Flags: ALLOC + WRITE

Sections

- `.rodata`:
  - This section holds read-only data.
  - Type: PROGBITS
  - Flags: ALLOC
- `.bss` :
  - This section holds data with no initial values. The system will initialize the data to zero when the program begins to run.
  - Type: NOBITS
  - Flags: ALLOC + WRITE

Sections

- `.rel.text`, `.rel.data`, and `.rel.rodata`:
  - These contain the relocation information for the corresponding text or data sections.
  - Type: REL
  - Flags: ALLOC is turned on if the file has a loadable segment that includes relocation.
- `.symtab`:
  - This section hold a symbol table.
- `.strtab`:
  - This section holds strings.

Sections

- `.init`:
  - This section holds executable instructions that contribute to the process initialization code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- `.fini`:
  - This section hold executable instructions that contribute to the process termination code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- Programming language specific.
  - C does not need these two sections. However, C++ needs them.

Sections

- `.interp`:
  - This section holds the pathname of a program interpreter.
  - Type: ALLOC
  - Flags: PROGBITS
  - If this section is present, rather than running the program directly, the system runs the interpreter and passes it the ELF file as an argument.
  - This facility runs non-text programs.
  - In practice, this is used to run the run-time dynamic linker to load the program and to link in any required shared libraries.
    - Variants of "`ld.so`" – the loader.

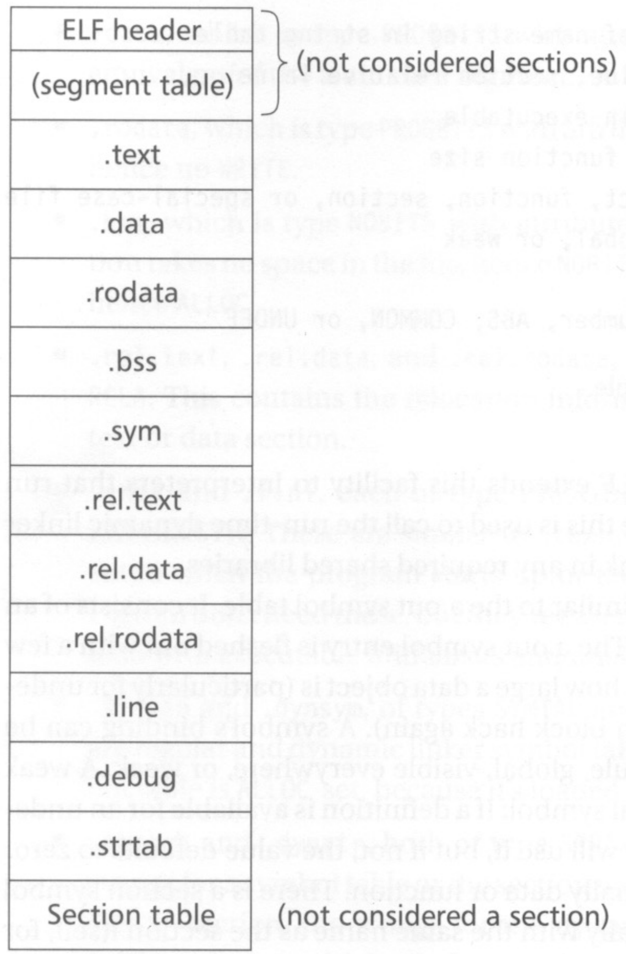
Sections

- `.debug`:
  - This section holds symbolic debugging information.
  - Type: PROGBIT
- `.line`:
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code (ever used gdb?)
  - Type: PROGBIT
- `.comment`
  - This section may store extra information.

Sections

- `.got`:
  - This section holds the global offset table.
    - Crucial for shared library.
  - Type: PROGBIT
- `.plt`:
  - This section holds the procedure linkage table.
  - Type: PROGBIT
- `.note`:
  - This section contains some extra information.

A typical relocatable file



String Table

- String table sections hold null-terminated character sequences, commonly called strings.
- The object file uses these strings to represent symbol and section names.
- We use an index into the string table section to reference a string.
- Separating symbol names from symbol tables frees us from any length limitation.

Symbol Table

- An array that holds information needed to locate and relocate a program’s symbolic definition and references.

- A symbol table index is a subscript into this array.

Symbol Table

```
typedef struct
{
  Elf32_Word    st_name;                /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;               /* Symbol value */
  Elf32_Word    st_size;                /* Symbol size */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf32_Section st_shndx;               /* Section index */
} Elf32_Sym;
```

The section relative in which the symbol is defined. (e.g., the function entry points are defined relative to .text)

An example of a symbol table

```
[wongwf@localhost ~]$ cat t.c
#include <stdio.h>

int example_of_global_var = 0x123456;

main()
{
    printf("Hello world -- %d\n", example_of_global_var);
}
```

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS t.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 example_of_global_var
    10: 0000000000000000    29 FUNC    GLOBAL DEFAULT    1 main
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

.symtab vs .dynsym

- Sharable objects (libraries) and dynamic executables has two distinct symbol tables
  - .dynsym $\subseteq$ .symtab
- .symtab is the full symbol table
  - Needed at static link time
  - Not all info needed at runtime hence not allocated in process memory
- .dynsym is the subset of .symtab needed at runtime
  - For runtime linking, loading and debugging

Symbol Lookup

- Two ways to lookup:
  - If you know the index, then use index the symbol table
  - If you only have a name, then you need the .hash or .gnu.hash sections
    - If `symbol_table[hash_table[hash(symbol_name)]] == symbol_name` then a hit (ok, more complicated than this…)

The four .gnu.hash sections

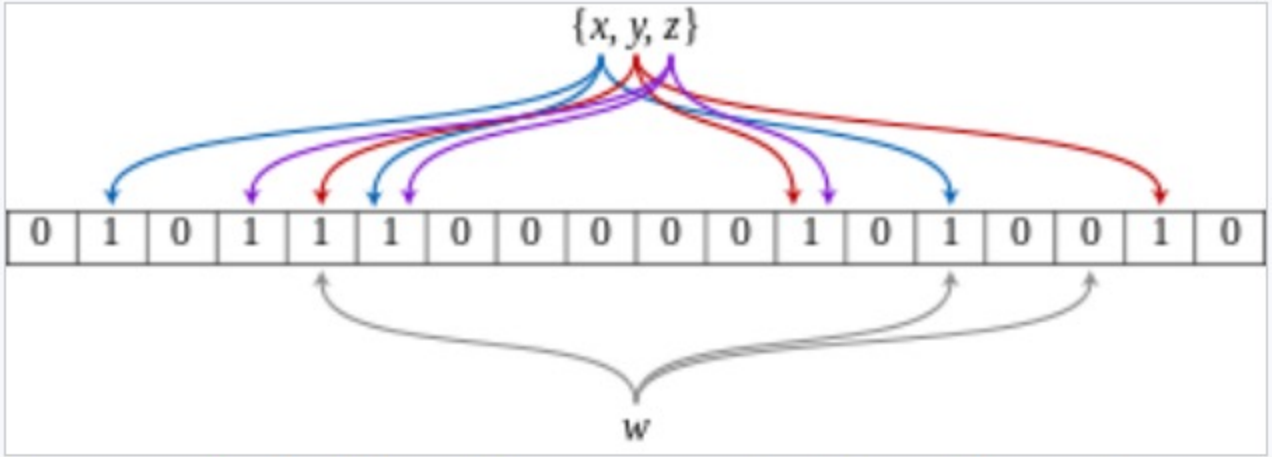
- Header: four 32-bit words
  - **`nbuckets`**: number of hash buckets
  - **`symndx`**: number of symbols of the dynamic symbol table that has been hashed
    - .dynsym may still contain other symbols not hashed
  - **`maskwords`**: number of words in the Bloom filter section
  - **`shift2`** – a shift count used in the Bloom filter
- Bloom Filter
- Hash Buckets
- Hash Values

Bloom Filters

- A Bloom filter probabilistically tests whether an element is a member of a set.
- False positive matches are possible, but false negatives are not.
  - A query returns either “possibly in set” or “definitely not in set”.
- Elements can be added to the set, but not removed.
- The more elements that are added to the set, the larger the probability of false positives.

- Wikipedia

Bloom Filter Example



An example of a Bloom filter, representing the set {x, y, z}. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set {x, y, z}, because it hashes to one bit-array position containing 0. For this figure, m = 18 and k = 3.

Wikipedia

GNU Hash

- Uses k=2 Bloom filter

```
static uint_fast32_t
dl_new_hash (const char *s)
{
        uint_fast32_t h = 5381;

        for (unsigned char c = *s; c != '\0'; c = *++s)
                h = h * 33 + c;

        return h & 0xffffffff;
}
```

  - `H1 = dl_new_hash(symbol_name)`
  - `H2 = H1 >> shift2`

C = size of one mask word in bits

  - `N = ((H1 / C) % maskwords)`
  - `BITMASK = (1 << (H1 % C)) | (1 << (H2 % C))`
  - `bloom[N] |= BITMASK`
  - Test: `(bloom[N] & BITMASK) == BITMASK`

Hash buckets

- An array where each entry N is the lowest index into the dynamic symbol table for which:

```
(dl_new_hash(symname) % nbuckets) == N
```

- **`dynsym[buckets[N]]`** is the first symbol in the hash chain that will contain the desired symbol if it exists.

High level walkthrough - 1

- Compute **`H1`**, a 32 or 64 bit (depending on the ELF type) hash value for the symbol name (a C string)
- Use part of **`H1`** to be another hash **`H2`**
- Use H1 to compute **`N`**, an index into the Bloom array
- Create a BITMASK – where there is two 1's at (**`H1 % C`**) and (**`H2 % C`**)
- Test if **`bloom`**[**`N`**] has two 1's at BITMASK positions
  - If false, we can confirm symbol not found

Hash values

- Last of the 4 parts of .gnu.hash
- One entry for every (hashed) symbol of .dynsym.
- The top 31 bits of each entry contains the top 31 bits of the corresponding symbol's hash value.
- The least significant bit is used as a stopper bit.
  - It is set to 1 when a symbol is the last symbol in a given hash chain.

Code walkthrough

Assume…

```
typedef struct {
        const char      *os_dynstr;      /* Dynamic string table */
        Sym             *os_dynsym;      /* Dynamic symbol table */
        Word            os_nbuckets;     /* # hash buckets */
        Word            os_symndx;       /* Index of 1st dynsym in hash */
        Word            os_maskwords_bm; /* # Bloom filter words, minus 1 */
        Word            os_shift2;       /* Bloom filter hash shift */
        const BloomWord *os_bloom;       /* Bloom filter words */
        const Word      *os_buckets;     /* Hash buckets */
        const Word      *os_hashval;     /* Hash value array */
} obj_state_t;
```

https://blogs.oracle.com/ali/entry/gnu_hash_elf_sections

Code walkthrough

```
Sym *
symhash(obj_state_t *os, const char *symname)
{
        Word            c;
        Word            h1, h2;
        Word            n;
        Word            bitmask;
        const Sym       *sym;
        Word            *hashval;

        /*
         * Hash the name, generate the "second" hash
         * from it for the Bloom filter.
         */
        h1 = dl_new_hash(symname);
        h2 = h1 >> os->os_shift2;

        /* Test against the Bloom filter */
        c = sizeof (BloomWord) * 8;
        n = (h1 / c) & os->os_maskwords_bm;
        bitmask = (1 << (h1 % c)) | (1 << (h2 % c));
        if ((os->os_bloom[n] & bitmask) != bitmask)
                return (NULL);
```

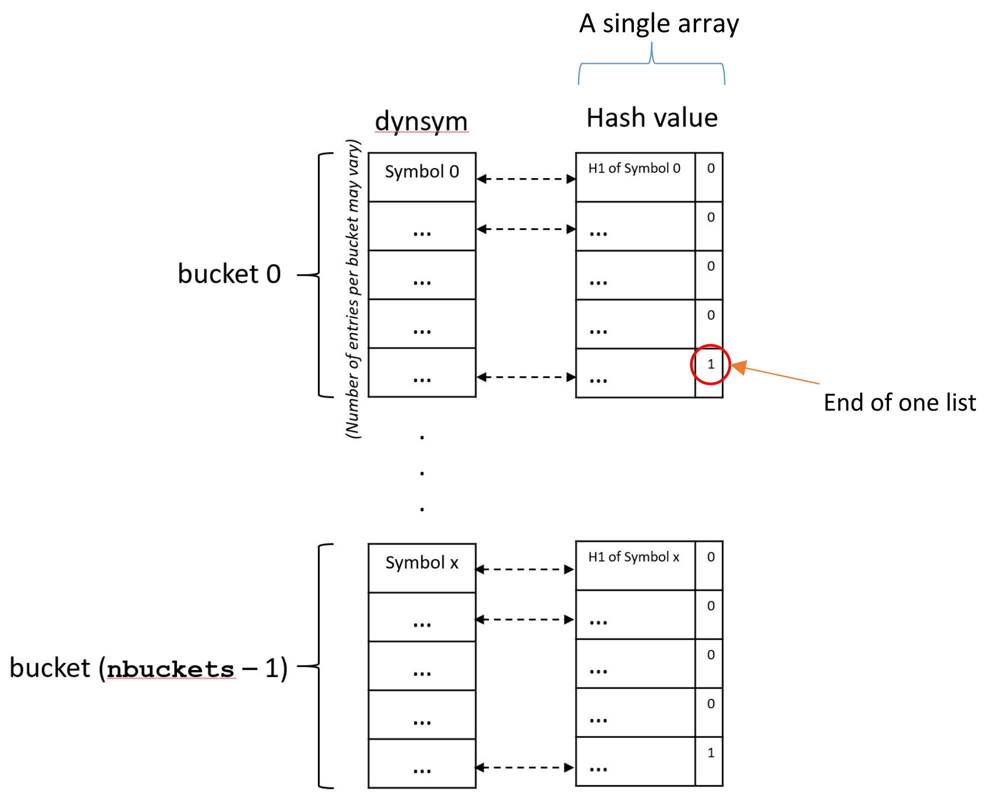
If Bloom test fails, for sure not found.

High level walkthrough - 2

- If it is true, we still need to check further coz Bloom filters can give false positives
- Dynamic symbol table organized by hash buckets
- Compute `n` – index into the hash bucket

Dynamic symbol table hash buckets

- Scan list of hash value
- If match found, compare symbol name string



Code walkthrough

```
/* Locate the hash chain, and corresponding hash value element */
n = os->os_buckets[h1 % os->os_nbuckets];
if (n == 0)    /* Empty hash chain, symbol not present */
        return (NULL);
sym = &os->os_dynsym[n];
hashval = &os->os_hashval[n - os->os_symndx];

/*
 * Walk the chain until the symbol is found or
 * the chain is exhausted.
 */
for (h1 &= ~1; 1; sym++) {
        h2 = *hashval++;

        /*
         * Compare the strings to verify match. Note that
         * a given hash chain can contain different hash
         * values. We'd get the right result by comparing every
         * string, but comparing the hash values first lets us
         * screen obvious mismatches at very low cost and avoid
         * the relatively expensive string compare.
         *
         * We are intentionally glossing over some things here:
         *
         *    -  We could test sym->st_name for 0, which indicates
         *       a NULL string, and avoid a strcmp() in that case.
         *
         *    - The real runtime linker must also take symbol
         *      versioning into account. This is an orthogonal
         *      issue to hashing, and is left out of this
         *      example for simplicity.
         *
         * A real implementation might test (h1 == (h2 & ~1), and then
         * call a (possibly inline) function to validate the rest.
         */
        if ((h1 == (h2 & ~1)) &&
            !strcmp(symname, os->os_dynstr + sym->st_name))
                return (sym);

        /* Done if at end of chain */
        if (h2 & 1)
                break;
}

/* This object does not have the desired symbol */
return (NULL);
```

A small digression: Name Mangling

- Names in the symbol tables are often not the same names used in the source code
- Three reasons:
  - Avoid name collision
  - Name overloading
  - Type checking
- Unfortunately, no standard – every compiler does its own thing

```
$ c++filt _ZNK3MapI10StringName3RefI8GDScriptE10ComparatorIS0_E16DefaultAllocatorE3hasERKS0_
Map<StringName, Ref<GDScript>, Comparator<StringName>, DefaultAllocator>::has(StringName const&) const
```

GNU C++ example - Wikipedia

Relocation Table

- Relocation is the process of connecting symbolic references with symbolic definitions.
- Relocatable files must have information that describes how to modify their section contents.
- A relocation table consists on many relocation structures.
- Essentially "determine the value of X, and put that value into the binary at offset Y"

Relocation Structure

- **`r_offset`**:
  - This field gives the location at which to apply the relocation.
  - For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affect by the relocation.
  - For an executable file and shared object, the value is the virtual address of the storage unit affected by the relocation.

Relocation Structure

- **`r_info`**:
  - This field gives both the symbol table index with respect to which the relocation must be made and the type of relocation to apply.
- **`r_addend`**:
  - This field specifies a constant addend used to compute the value to be stored into the relocation field.

Example

```
Relocation section '.rela.text' at offset 0x5d0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000006  000900000002 R_X86_64_PC32     0000000000000000 example_of_global_var - 4
00000000000d  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
000000000017  000b00000002 R_X86_64_PC32     0000000000000000 printf - 4
```

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # a <main+0xa>
   a:   89 c6                   mov    %eax,%esi
   c:   bf 00 00 00 00          mov    $0x0,%edi
  11:   b8 00 00 00 00          mov    $0x0,%eax
  16:   e8 00 00 00 00          callq  1b <main+0x1b>
  1b:   5d                      pop    %rbp
  1c:   c3                      retq
```

"In the final binary, patch the value at offset 0x6 in this object file with the address of symbol example_of_a_global_var"

Executable Files

- An executable file usually has only a few segments. E.g.,
  - A read-only one for the code.
  - A read-only one for read-only data.
  - A read/write one for read/write data.
- All of the loadable sections are packed into the appropriate segments so that the system can map the file with just one or two operations.
  - E.g., If there is a `.init` and `.fini` sections, those sections will be put into the read-only text segment.

Program 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;
```

The Types in Program Header

- PT_LOAD: This segment is a loadable segment.

- PT_DYNAMIC: This array element specifies dynamic linking information.

- PT_INTERP: This element specified the location and size of a null-terminated path name to invoke as an interpreter.

Executable File Example

	File offset	*Load address*	*Type header*
ELF header	0	0x8000000	
Program header	0x40	0x8000040	
Read-only text (size 0x4500)	0x100	0x8000100	LOAD, read/execute
Read/write data (file size 0x2200, memory size 0x3500)	0x4600	0x8005600	LOAD, read/write/ execute
Nonloadable information and optional section headers			

FIGURE 3.16 • ELF loadable segments.

An Example C Program

```
[wongwf@localhost ~]$ cat t.c
#include <stdio.h>

int example_of_global_var = 0x123456;

main()
{
    printf("Hello world -- %d\n", example_of_global_var);
}
```

ELF Header Information

```
[wongwf@localhost ~]$ objdump -f a.out

a.out:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400440
```

Program Header

```
[wongwf@localhost ~]$ objdump -p a.out

a.out:     file format elf64-x86-64

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
         filesz 0x000000000000071c memsz 0x000000000000071c flags r-x
    LOAD off    0x0000000000000e10 vaddr 0x0000000000600e10 paddr 0x0000000000600e10 align 2**21
         filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
 DYNAMIC off    0x0000000000000e28 vaddr 0x0000000000600e28 paddr 0x0000000000600e28 align 2**3
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
    NOTE off    0x0000000000000254 vaddr 0x0000000000400254 paddr 0x0000000000400254 align 2**2
         filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
EH_FRAME off    0x00000000000005f4 vaddr 0x00000000004005f4 paddr 0x00000000004005f4 align 2**2
         filesz 0x0000000000000034 memsz 0x0000000000000034 flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
   RELRO off    0x0000000000000e10 vaddr 0x0000000000600e10 paddr 0x0000000000600e10 align 2**0
         filesz 0x00000000000001f0 memsz 0x00000000000001f0 flags r--
```

Dynamic Section

```
Dynamic section:
  NEEDED               libc.so.6
  INIT                 0x00000000004003e0
  FINI                 0x00000000004005c4
  INIT_ARRAY           0x0000000000600e10
  INIT_ARRAYSZ         0x0000000000000008
  FINI_ARRAY           0x0000000000600e18
  FINI_ARRAYSZ         0x0000000000000008
  GNU_HASH             0x0000000000400298
  STRTAB               0x0000000000400318
  SYMTAB               0x00000000004002b8
  STRSZ                0x000000000000003f
  SYMENT               0x0000000000000018
  DEBUG                0x0000000000000000
  PLTGOT               0x0000000000601000
  PLTRELSZ             0x0000000000000048
  PLTREL               0x0000000000000007
  JMPREL               0x0000000000400398
  RELA                 0x0000000000400380
  RELASZ               0x0000000000000018
  RELAENT              0x0000000000000018
  VERNEED              0x0000000000400360
  VERNEEDNUM           0x0000000000000001
  VERSYM               0x0000000000400358

Version References:
  required from libc.so.6:
    0x09691a75 0x00 02 GLIBC_2.2.5
```

Need to link this shared library for printf()

Section Header

```
[wongwf@localhost ~]$ objdump -h a.out

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000400274  0000000000400274  00000274  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.hash     0000001c  0000000000400298  0000000000400298  00000298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynsym       00000060  00000000004002b8  00000000004002b8  000002b8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynstr       0000003f  0000000000400318  0000000000400318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version  00000008  0000000000400358  0000000000400358  00000358  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version_r 00000020  0000000000400360  0000000000400360  00000360  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rela.dyn     00000018  0000000000400380  0000000000400380  00000380  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.plt     00000048  0000000000400398  0000000000400398  00000398  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000001a  00000000004003e0  00000000004003e0  000003e0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000040  0000000000400400  0000000000400400  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         00000184  0000000000400440  0000000000400440  00000440  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         00000009  00000000004005c4  00000000004005c4  000005c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .rodata       00000023  00000000004005d0  00000000004005d0  000005d0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .eh_frame_hdr 00000034  00000000004005f4  00000000004005f4  000005f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame     000000f4  0000000000400628  0000000000400628  00000628  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .init_array   00000008  0000000000600e10  0000000000600e10  00000e10  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 18 .fini_array   00000008  0000000000600e18  0000000000600e18  00000e18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 19 .jcr          00000008  0000000000600e20  0000000000600e20  00000e20  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 20 .dynamic      000001d0  0000000000600e28  0000000000600e28  00000e28  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .got          00000008  0000000000600ff8  0000000000600ff8  00000ff8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .got.plt      00000030  0000000000601000  0000000000601000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .data         00000008  0000000000601030  0000000000601030  00001030  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          00000008  0000000000601038  0000000000601038  00001038  2**2
                  ALLOC
 25 .comment      0000002c  0000000000000000  0000000000000000  00001038  2**0
                  CONTENTS, READONLY
```

Symbol Table

```
[wongwf@localhost ~]$ objdump -t a.out

a.out:     file format elf64-x86-64

SYMBOL TABLE:
0000000000400238 l    d  .interp	0000000000000000              .interp
0000000000400254 l    d  .note.ABI-tag	0000000000000000              .note.ABI-tag
0000000000400274 l    d  .note.gnu.build-id	0000000000000000              .note.gnu.build-id
0000000000400298 l    d  .gnu.hash	0000000000000000              .gnu.hash
00000000004002b8 l    d  .dynsym	0000000000000000              .dynsym
0000000000400318 l    d  .dynstr	0000000000000000              .dynstr
0000000000400358 l    d  .gnu.version	0000000000000000              .gnu.version
0000000000400360 l    d  .gnu.version_r	0000000000000000              .gnu.version_r
0000000000400380 l    d  .rela.dyn	0000000000000000              .rela.dyn
0000000000400398 l    d  .rela.plt	0000000000000000              .rela.plt
00000000004003e0 l    d  .init	0000000000000000              .init
0000000000400400 l    d  .plt	0000000000000000              .plt
0000000000400440 l    d  .text	0000000000000000              .text
00000000004005c4 l    d  .fini	0000000000000000              .fini
00000000004005d0 l    d  .rodata	0000000000000000              .rodata
00000000004005f4 l    d  .eh_frame_hdr	0000000000000000              .eh_frame_hdr
0000000000400628 l    d  .eh_frame	0000000000000000              .eh_frame
0000000000600e10 l    d  .init_array	0000000000000000              .init_array
0000000000600e18 l    d  .fini_array	0000000000000000              .fini_array
0000000000600e20 l    d  .jcr	0000000000000000              .jcr
0000000000600e28 l    d  .dynamic	0000000000000000              .dynamic
0000000000600ff8 l    d  .got	0000000000000000              .got
0000000000601000 l    d  .got.plt	0000000000000000              .got.plt
0000000000601030 l    d  .data	0000000000000000              .data
0000000000601038 l    d  .bss	0000000000000000              .bss
0000000000000000 l    d  .comment	0000000000000000              .comment
0000000000000000 l    df *ABS*	0000000000000000              crtstuff.c
0000000000600e20 l     O .jcr	0000000000000000              __JCR_LIST__
0000000000400470 l     F .text	0000000000000000              deregister_tm_clones
00000000004004a0 l     F .text	0000000000000000              register_tm_clones
00000000004004e0 l     F .text	0000000000000000              __do_global_dtors_aux
0000000000601038 l     O .bss	0000000000000001              completed.6337
0000000000600e18 l     O .fini_array	0000000000000000              __do_global_dtors_aux_fini_array_entry
0000000000400500 l     F .text	0000000000000000              frame_dummy
0000000000600e10 l     O .init_array	0000000000000000              __frame_dummy_init_array_entry
0000000000000000 l    df *ABS*	0000000000000000              t.c
0000000000000000 l    df *ABS*	0000000000000000              crtstuff.c
0000000000400718 l     O .eh_frame	0000000000000000              __FRAME_END__
0000000000600e20 l     O .jcr	0000000000000000              __JCR_END__
0000000000000000 l    df *ABS*	0000000000000000
0000000000600e18 l       .init_array	0000000000000000              __init_array_end
0000000000600e28 l     O .dynamic	0000000000000000              _DYNAMIC
0000000000600e10 l       .init_array	0000000000000000              __init_array_start
0000000000601000 l     O .got.plt	0000000000000000              _GLOBAL_OFFSET_TABLE_
00000000004005c0 g     F .text	0000000000000002              __libc_csu_fini
0000000000601034 g     O .data	0000000000000004              example_of_global_var
0000000000000000  w      *UND*	0000000000000000              _ITM_deregisterTMCloneTable
0000000000601030  w      .data	0000000000000000              data_start
0000000000601038 g       .data	0000000000000000              _edata
00000000004005c4 g     F .fini	0000000000000000              _fini
0000000000000000       F *UND*	0000000000000000              printf@@GLIBC_2.2.5
0000000000000000       F *UND*	0000000000000000              __libc_start_main@@GLIBC_2.2.5
0000000000601030 g       .data	0000000000000000              __data_start
0000000000000000  w      *UND*	0000000000000000              __gmon_start__
00000000004005d8 g     O .rodata	0000000000000000              .hidden __dso_handle
00000000004005d0 g     O .rodata	0000000000000004              _IO_stdin_used
0000000000400550 g     F .text	0000000000000065              __libc_csu_init
0000000000601040 g       .bss	0000000000000000              _end
0000000000400440 g     F .text	0000000000000000              _start
0000000000601038 g       .bss	0000000000000000              __bss_start
0000000000400530 g     F .text	000000000000001d              main
0000000000000000  w      *UND*	0000000000000000              _Jv_RegisterClasses
0000000000601038 g     O .data	0000000000000000              .hidden __TMC_END__
0000000000000000  w      *UND*	0000000000000000              _ITM_registerTMCloneTable
00000000004003e0 g     F .init	0000000000000000              _init
```

Dynamic Symbol Table

```
[wongwf@localhost ~]$ objdump -T a.out

a.out:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND*  0000000000000000  GLIBC_2.2.5 printf
0000000000000000      DF *UND*  0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000  w   D  *UND*  0000000000000000              __gmon_start__
```

Dynamic Relocation Table

```
[wongwf@localhost ~]$ objdump -R a.out

a.out:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601018 R_X86_64_JUMP_SLOT  printf
0000000000601020 R_X86_64_JUMP_SLOT  __libc_start_main
0000000000601028 R_X86_64_JUMP_SLOT  __gmon_start__
```

Allocation in Linking

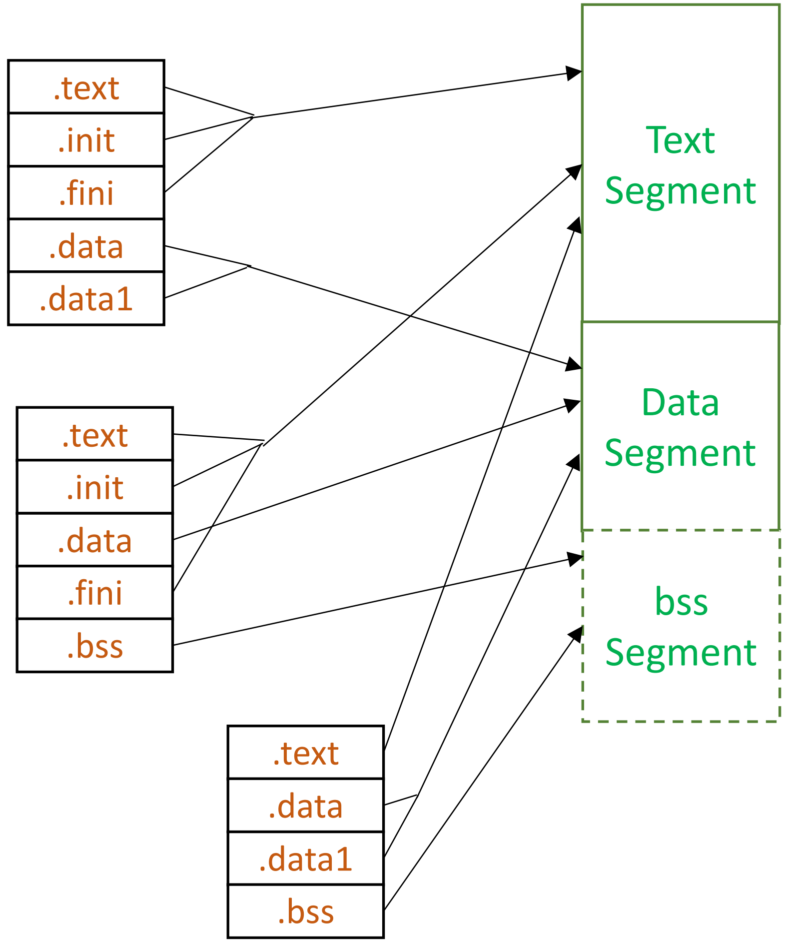
Need for allocation

- Each object file compiled differently with no idea about the others
- All objects in a given object file assumed to start at address 0
- Need to combine into a single executable

Combining ELF object files

- Multiple segments needs to be combined

- Linker will collect the sections and place them into the segments of the final executable image plus write the program header



You can control it!

- Linkers can be controlled via link scripts
- In GNU ld linker, you can use the “-T” option

“Start .text at virtual address 0x10000”

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

“Start .data followed by .bss at virtual address 0x8000000”

A very simple example

Dynamic Linking

Motivation

- Shared libraries: only one copy of commonly used routines for all code in a system
  - Easy to maintain
  - Save space
- External symbols referenced in user code and defined in a shared library are resolved by the loader at load time.

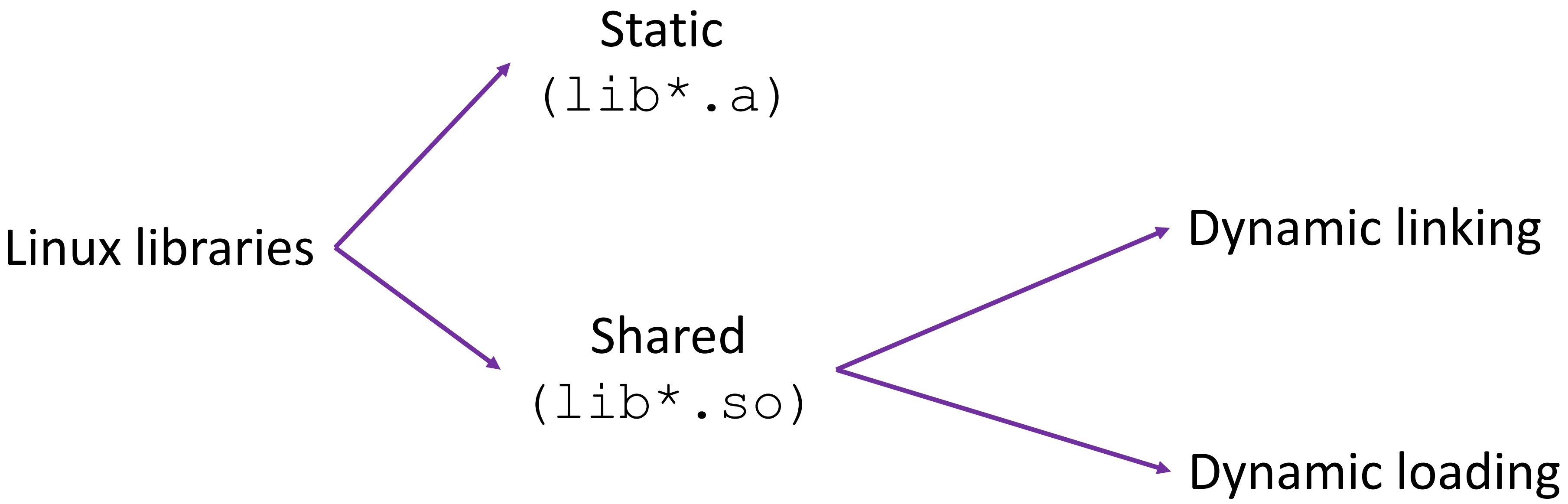
Advantages

- Faster load time: shared library code may already be in memory
- Better run-time performance: less likely to page out frequently used code
- Easier for shared libraries to be updated

Disadvantages

- Need “glue code”
- Reduced locality of references
- May impact paging
- Changes in libraries may break some applications

The Linux Library Hierarchy



Static Libraries

- Simply a concatenation of object files with simple headers

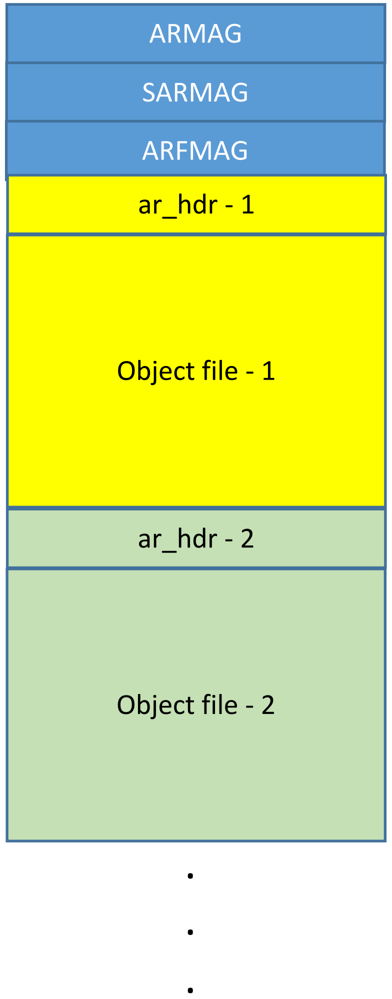
- In Linux, it is created using the "**`ar`**" utility and has the file extension of "**`.a`**"

Linux static library format

```
/* Archive files start with the ARMAG identifying string.  Then follows a
   `struct ar_hdr', and as many bytes of member file data as its `ar_size'
   member indicates, for each member file.  */

#define ARMAG   "!<arch>\n"     /* String that begins an archive file.  */
#define SARMAG  8               /* Size of that string.  */
#define ARFMAG  "`\n"           /* String in ar_fmag at end of each header.  */

struct ar_hdr
  {
    char ar_name[16];           /* Member file name, sometimes / terminated. */
    char ar_date[12];           /* File date, decimal seconds since Epoch.  */
    char ar_uid[6], ar_gid[6];  /* User and group IDs, in ASCII decimal.  */
    char ar_mode[8];            /* File mode, in ASCII octal.  */
    char ar_size[10];           /* File size, in ASCII decimal.  */
    char ar_fmag[2];            /* Always contains ARFMAG.  */
  };
```



Basically…

If `libx.a` consists of `obj1.o`, `obj2.o`, …, `objk.o` concatenated

Then

```
gcc … libx.a
```

Is just

```
gcc … obj1.o obj2.o … objk.o
```

Dynamic linking: issues to resolve

- Allocate and load the segments found in the shared libraries
- Perform relocation
  - Recall: relocations are entries in binaries that are left to be filled in later
- Two solutions:
  - Load-time relocation
  - Position-independence code (PIC)

Key Data Structures

- Procedure Linkage Table (PLT)
  - One entry for each external function reference.
  - All calls to the corresponding external function routed through this entry.
  - Easier to patch with actual address
- Global Offset Table (GOT)
  - All external global addresses.

The special three entries of the GOT

- GOT[0]: address of the program's `.dynamic` segment
- GOT[1]: pointer to a linked list of nodes corresponding to the symbol tables for each shared library linked with the program
- GOT[2]: address of the symbol resolution function within the dynamic linker
  - `/lib{64}/ld-linux{-x86-64}.so`

How it works (preparation - 1)

- Compiler ensures every call to the same dynamically linked routine will go to the same PLT entry
  - PLT[0] is reserved
- Each (64) bit PLT entry consists of:
  - `jmpq` `*GOT`[entry corresponding to the function]
    - Called a “trampoline”.
  - `pushq` <GOT entry number of corresponding function>
    - Starts from 0, i.e. GOT[3].
  - `jmp` PLT[0]
- `GOT`[entry corresponding to the function] = address of “`pushq`” *initially*

How it works (preparation - 2)

- `.rela.plt` section of executable containing the GOT has the relocation information to fix up the GOT at runtime

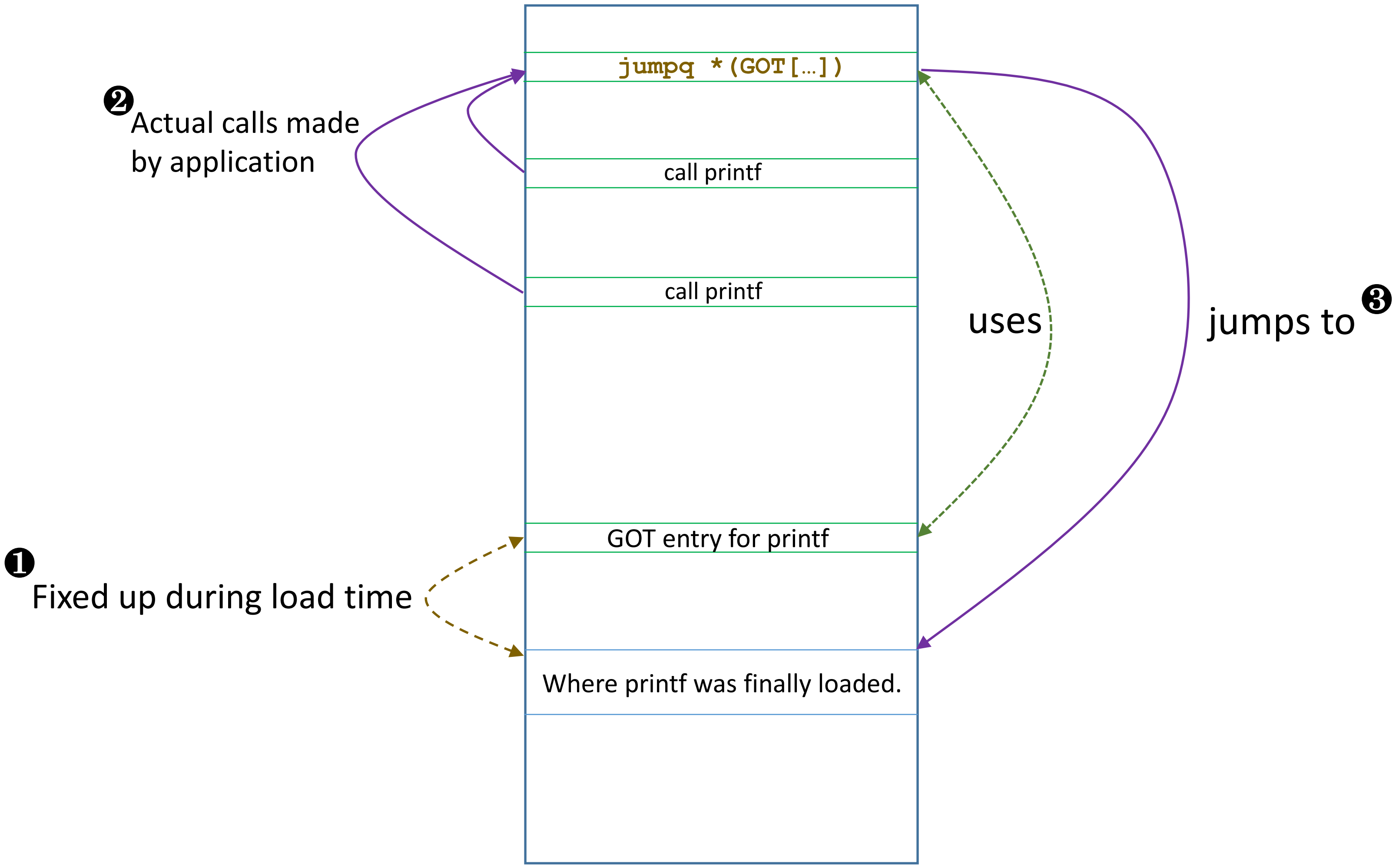
- Address of `.rela.plt` itself is found as an entry in `.dynamic`

- Two ways to do dynamic linking:
  - “BIND NOW”
  - Lazy binding

“BIND NOW”

- Find and load in all dynamic libraries
- Go through the main application’s GOT relocation entries and fix all entries
- After fixing up, `GOT`[entry corresponding to the function] = address of actual function in the process image
- Therefore, trampoline jump `jmpq *GOT`[entry corresponding to the function] will jump directly to the function
  - `pushq` and the second `jmp` never used

“BIND NOW”



jumpq *(GOT[…])
❷
Actual calls made
by application
call printf
call printf
uses
jumps to ❸
❶
Fixed up during load time
GOT entry for printf
Where printf was finally loaded.

Lazy Binding

- "BIND NOW" can slow down program start up
  - Define the environment variable **`LD_BIND_NOW`**
  - Many functions may not be used in a single execution anyway

- Solution: resolve binding only when we need it

- The reserved PLT[0] entry:
  - **`pushq GOT[1]`**
  - **`jmpq *(GOT[2])`**

GOT[2]

- Usually it is `_dl_runtime_resolve()`
  - In turn will call `_dl_fixup()`

- If profiling enabled, then `_dl_runtime_profile()`

- Assembly code in `glibc/dl-trampoline.S`

Being lazy - 1

- Recall **`GOT`**[entry corresponding to the function] = address of "**`pushq`**" *initially*

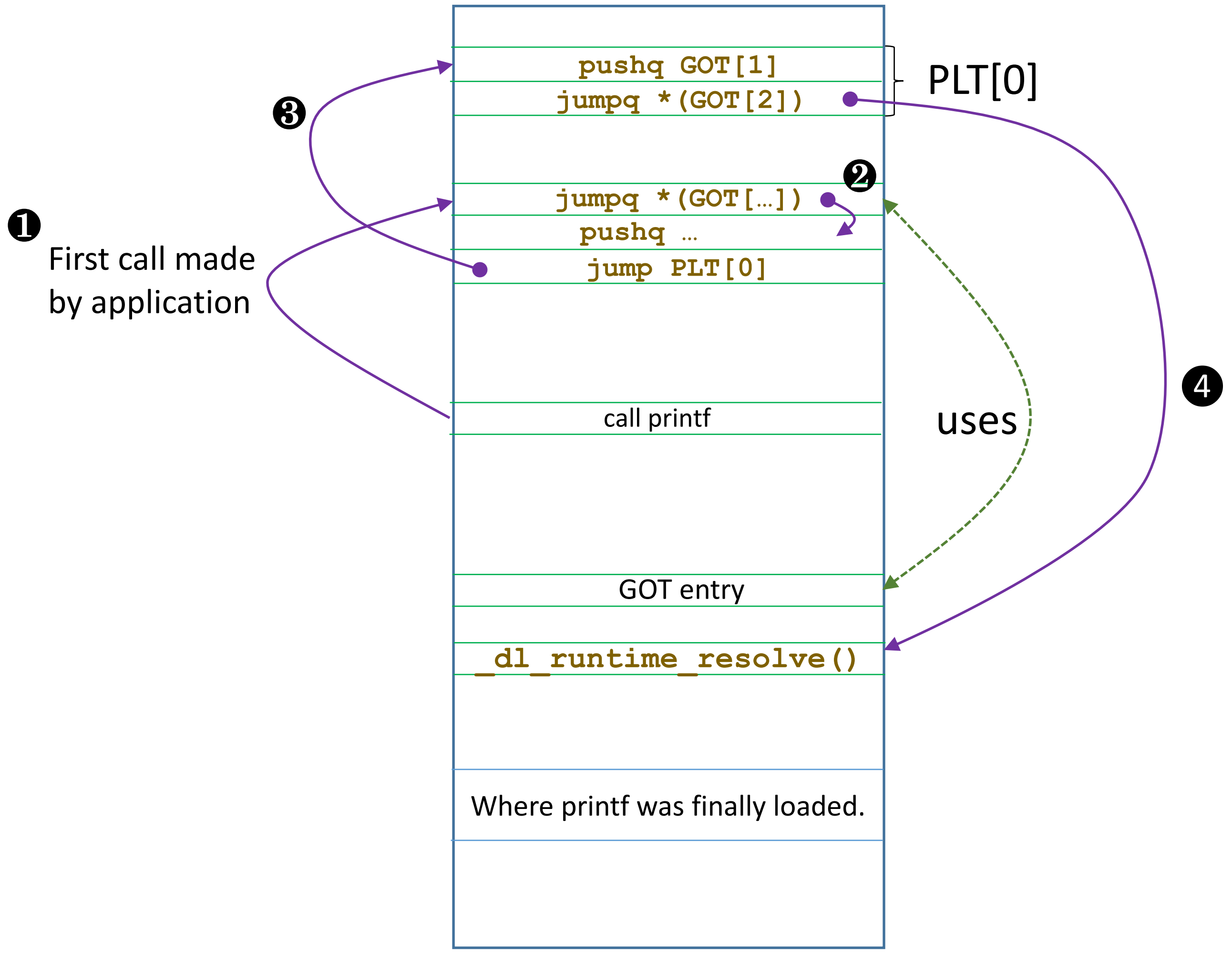
- Lazy = not immediately change GOT entry at load time

- First time the PLT function entry is called, it will do **`jumpq *(…)`** – jumps right back to the second instruction of the PLT entry, i.e. do:
  - **`pushq`** <GOT entry number of corresponding function>
  - **`jmp`** PLT[0]

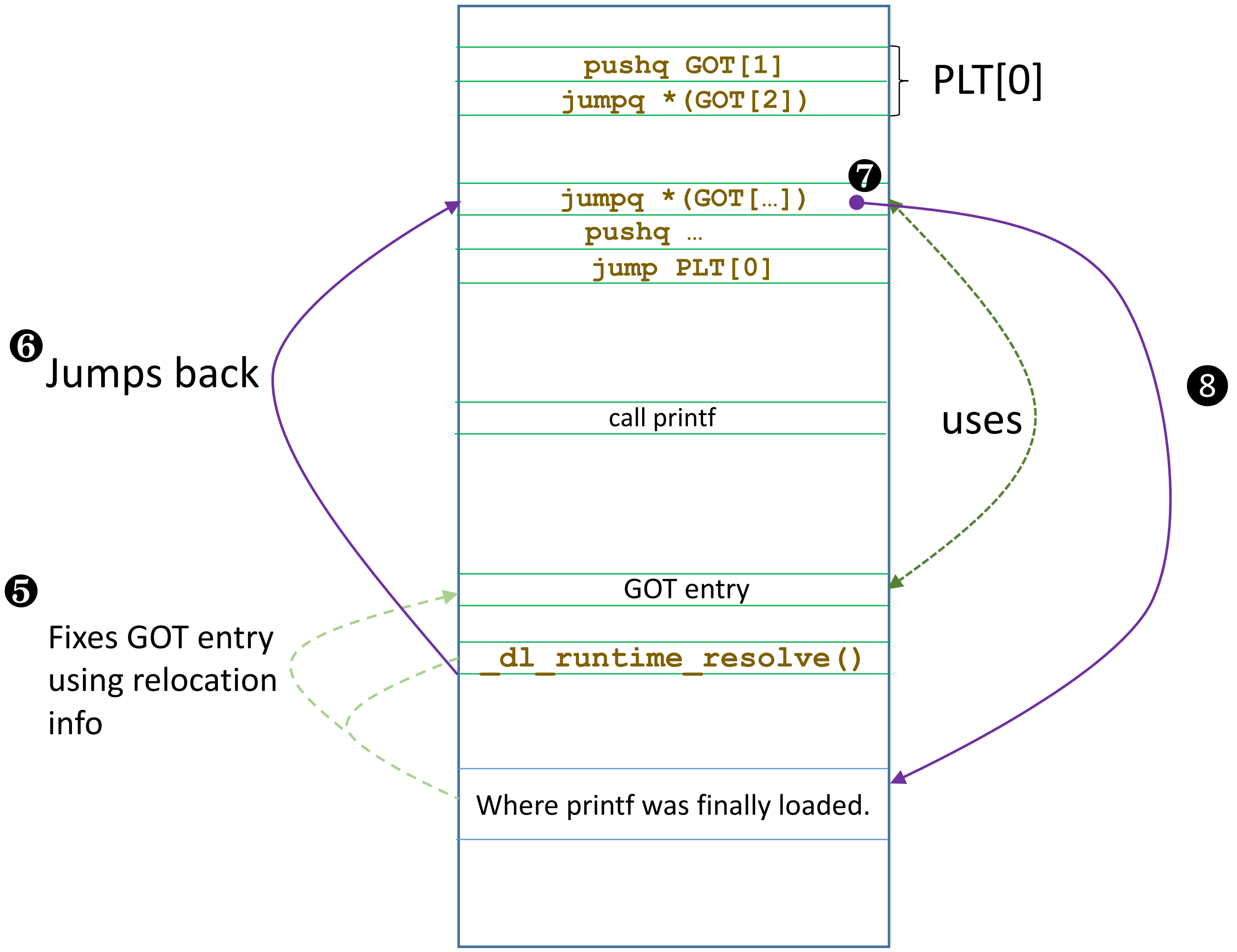
Being lazy - 2

- PLT[0] will jmp to `_dl_runtime_resolve()` of `/lib{64}/ld-linux{-x86-64}.so`
- `_dl_runtime_resolve()` will find final call site and using the relocation information patch the GOT entry
- Jumps back to PLT entry of the procedure. This time and all subsequent time, `jumpq *(…)` will jump using the patched GOT entry

Being Lazy - 3


pushq GOT[1]
jumpq *(GOT[2])
PLT[0]
❸
❷
jumpq *(GOT[…])
pushq …
jump PLT[0]
❶
First call made by application
call printf
uses
❹
GOT entry
_dl_runtime_resolve()
Where printf was finally loaded.

Being Lazy - 4


pushq GOT[1]
jumpq *(GOT[2])
PLT[0]
❼
jumpq *(GOT[…])
pushq …
jump PLT[0]
❻ Jumps back
call printf
uses
❽
GOT entry
❺
Fixes GOT entry using relocation info
_dl_runtime_resolve()
Where printf was finally loaded.

What about the shared objects themselves?

- Suppose a shared library itself uses a function in another?
- Shared libraries are also ELF
- They have their own PLT, GOT and relocation information embedded in the dynamic segment
- These too has to be resolved at runtime

Position Independent Code

- Problem: hard to fix system-wide where a shared object should reside
  - Any one can create shared library.
  - Address Space Randomization (ASR)
- 64-bit Linux solution: shared objects must be compiled with "`-fPIC`"
- PIC will work anywhere it is placed
  - All branching uses relative addressing
  - Local data no issue (relative to stack or base pointer)
  - Global data: use indirect loading via GOT
  - Global shared procedure: use PLT and GOT

(Just for fun) PIC example in 32 bit x86

```
#include <stdio.h>

int globvar = 42;

int foo(int arg)
{
    return globvar + arg;
}
```

C source

```
[CS5250]$ cc -S -m32 PIC-example.c
[CS5250]$ cat PIC-example.s
        .file   "PIC-example.c"
        .globl  globvar
        .data
        .align 4
        .type   globvar, @object
        .size   globvar, 4
globvar:
        .long   42
        .text
        .globl  foo
        .type   foo, @function
foo:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        movl    globvar, %edx
        movl    8(%ebp), %eax
        addl    %edx, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
        .section        .note.GNU-stack,"",@progbits
```

```
[CS5250]$ cc -c -m32 PIC-example.c
[CS5250]$ objdump -d PIC-example.o

PIC-example.o:     file format elf32-i386


Disassembly of section .text:

00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 15 00 00 00 00       mov    0x0,%edx
   9:   8b 45 08                mov    0x8(%ebp),%eax
   c:   01 d0                   add    %edx,%eax
   e:   5d                      pop    %ebp
   f:   c3                      ret
```

Linker/Loader to fill this in

(Just for fun) PIC example in 32 bit x86

```
[CS5250]$ cc -S -m32 -fPIC PIC-example.c
[CS5250]$ cat PIC-example.s
        .file   "PIC-example.c"
        .globl  globvar
        .data
        .align 4
        .type   globvar, @object
        .size   globvar, 4
globvar:
        .long   42
        .text
        .globl  foo
        .type   foo, @function
foo:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        call    __x86.get_pc_thunk.cx
        addl    $_GLOBAL_OFFSET_TABLE_, %ecx
        movl    globvar@GOT(%ecx), %eax
        movl    (%eax), %edx
        movl    8(%ebp), %eax
        addl    %edx, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .section        .text.__x86.get_pc_thunk.cx,"axG",@progbits,
        .globl  __x86.get_pc_thunk.cx
        .hidden __x86.get_pc_thunk.cx
        .type   __x86.get_pc_thunk.cx, @function
__x86.get_pc_thunk.cx:
.LFB1:
        .cfi_startproc
        movl    (%esp), %ecx
        ret
        .cfi_endproc
.LFE1:
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
        .section        .note.GNU-stack,"",@progbits
```

Use GOT to indirectly obtain variable's address

%ecx contains EIP after call

Linker/loader to fill these in

Move return address to %ecx

```
[CS5250]$ objdump -d PIC-example.o

PIC-example.o:     file format elf32-i386


Disassembly of section .text:

00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   e8 fc ff ff ff          call   4 <foo+0x4>
   8:   81 c1 02 00 00 00       add    $0x2,%ecx
   e:   8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  14:   8b 10                   mov    (%eax),%edx
  16:   8b 45 08                mov    0x8(%ebp),%eax
  19:   01 d0                   add    %edx,%eax
  1b:   5d                      pop    %ebp
  1c:   c3                      ret

Disassembly of section .text.__x86.get_pc_thunk.cx:

00000000 <__x86.get_pc_thunk.cx>:
   0:   8b 0c 24                mov    (%esp),%ecx
   3:   c3                      ret
```

(Just for fun) PIC made easier in x86-64

```
[CS5250]$ cc -S  -fPIC PIC-example.c
[CS5250]$ cat PIC-example.s
        .file   "PIC-example.c"
        .globl  globvar
        .data
        .align 4
        .type   globvar, @object
        .size   globvar, 4
globvar:
        .long   42
        .text
        .globl  foo
        .type   foo, @function
foo:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movq    globvar@GOTPCREL(%rip), %rax
        movl    (%rax), %edx
        movl    -4(%rbp), %eax
        addl    %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red H
        .section        .note.GNU-stack,"",@progb
```

New addressing mode in x86-64

```
[CS5250]$ cc -c  -fPIC PIC-example.c
[CS5250]$ objdump -d PIC-example.o

PIC-example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
   e:   8b 10                   mov    (%rax),%edx
  10:   8b 45 fc                mov    -0x4(%rbp),%eax
  13:   01 d0                   add    %edx,%eax
  15:   5d                      pop    %rbp
  16:   c3                      retq
```

Linker/Loader to fill this in

Dynamic Loading

Dynamic Loading API

- Loading a shared library under the control of the application

- Linux DL API
  - Must link with "`-ldl`" option

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

An example

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int
main(int argc, char **argv)
{
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen("libm.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    dlerror();    /* Clear any existing error */

    /* Writing: cosine = (double (*)(double)) dlsym(handle, "cos");
       would seem more natural, but the C99 standard leaves
       casting from "void *" to a function pointer undefined.
       The assignment used below is the POSIX.1-2003 (Technical
       Corrigendum 1) workaround; see the Rationale for the
       POSIX specification of dlsym(). */

    *(void **) (&cosine) = dlsym(handle, "cos");

    if ((error = dlerror()) != NULL)  {
        fprintf(stderr, "%s\n", error);
        exit(EXIT_FAILURE);
    }

    printf("%f\n", (*cosine)(2.0));
    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

Compile with:

```
gcc -rdynamic -o foo foo.c -ldl
```

End