

The link from object to executable

The role of the linker in a development toolchain
Peter Smith ACCU 2017

What are we covering today?

- What is a linker?
- Introduction to object files?
- Linking process
- Dynamic linking
- Advanced topics
- Concluding thoughts

My background with Linkers

- Worked in ARM's proprietary toolchain team since 2000
 - Focus on embedded systems
 - armcc, armasm, **armlink**, fromelf
 - SDT, ADS, RVCT, ARM Compiler (MDK, DS-5, mBed)
- Assigned to Linaro from 2016
 - Adding ARM support to the llvm linker **lld**

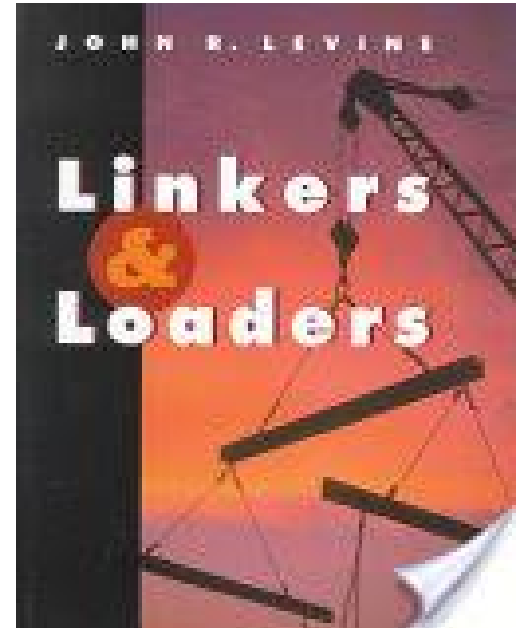
What is a linker?

What does a linker do exactly?

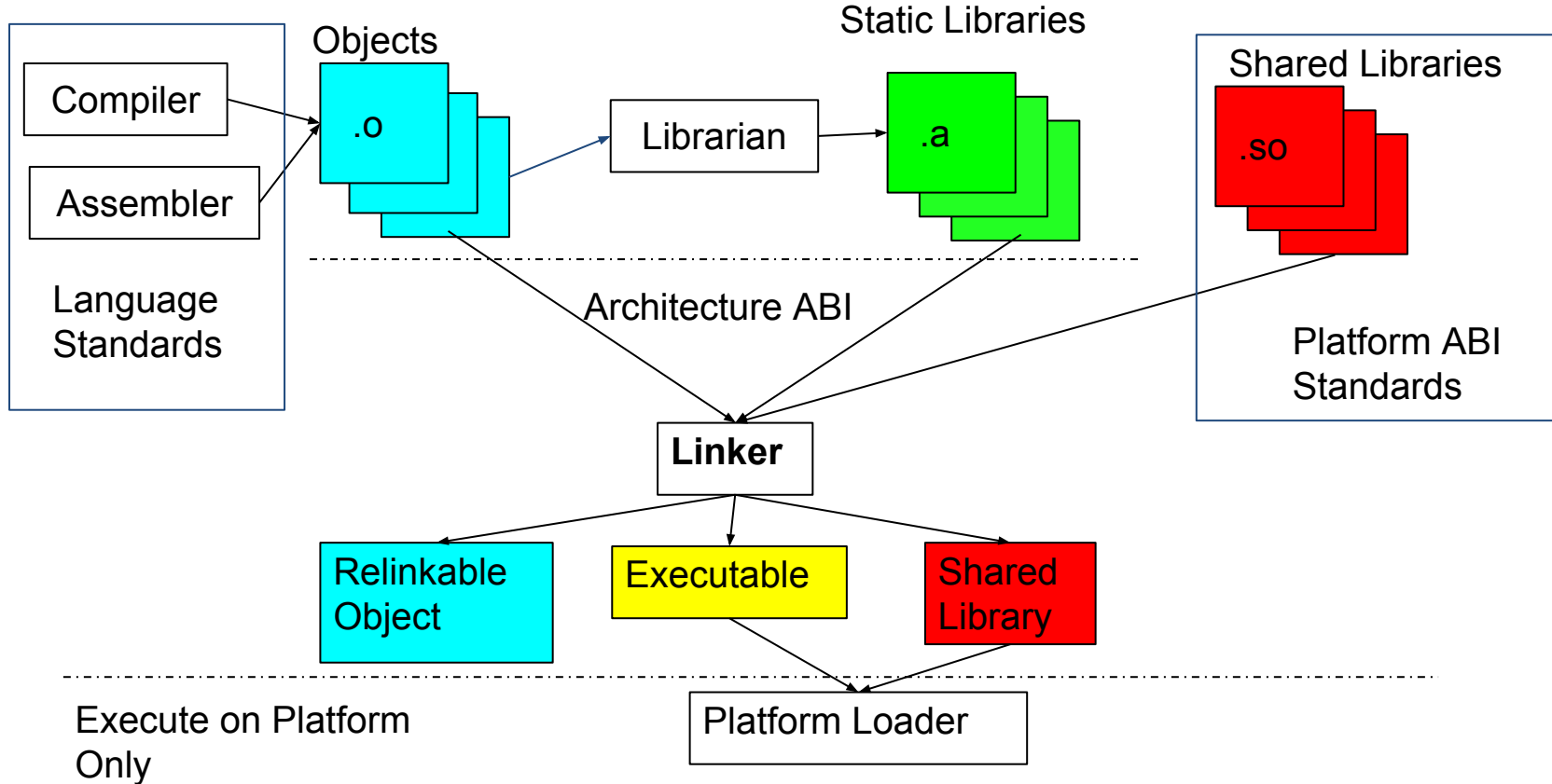
What role does it play in the toolchain

What is a Linker?

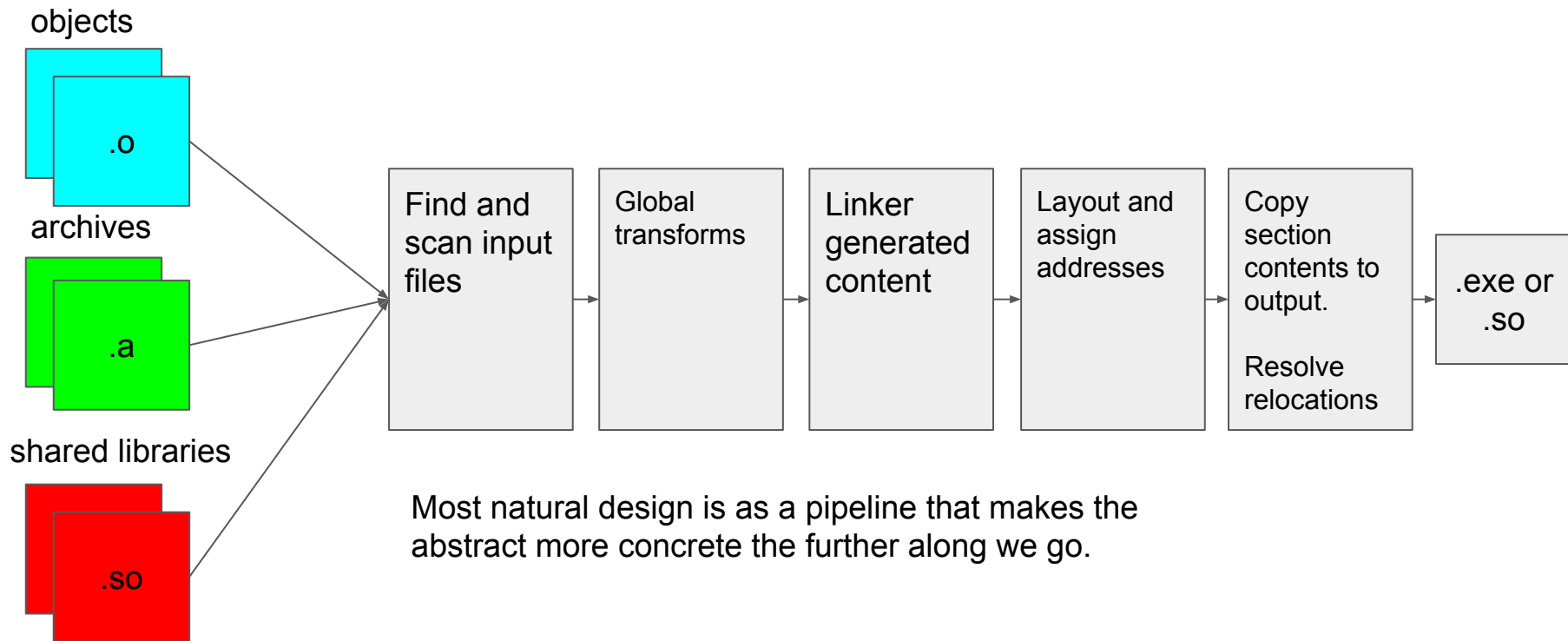
- Levine defines in Linkers and Loaders
 - “Binds more abstract names to more concrete names, which permits programmers to write code using more abstract names”
 - In general concrete representations are higher performance but less flexible
- In practice a tool that glues together the outputs of separate compilation into a single output



Role of a linker in the toolchain



Linker design

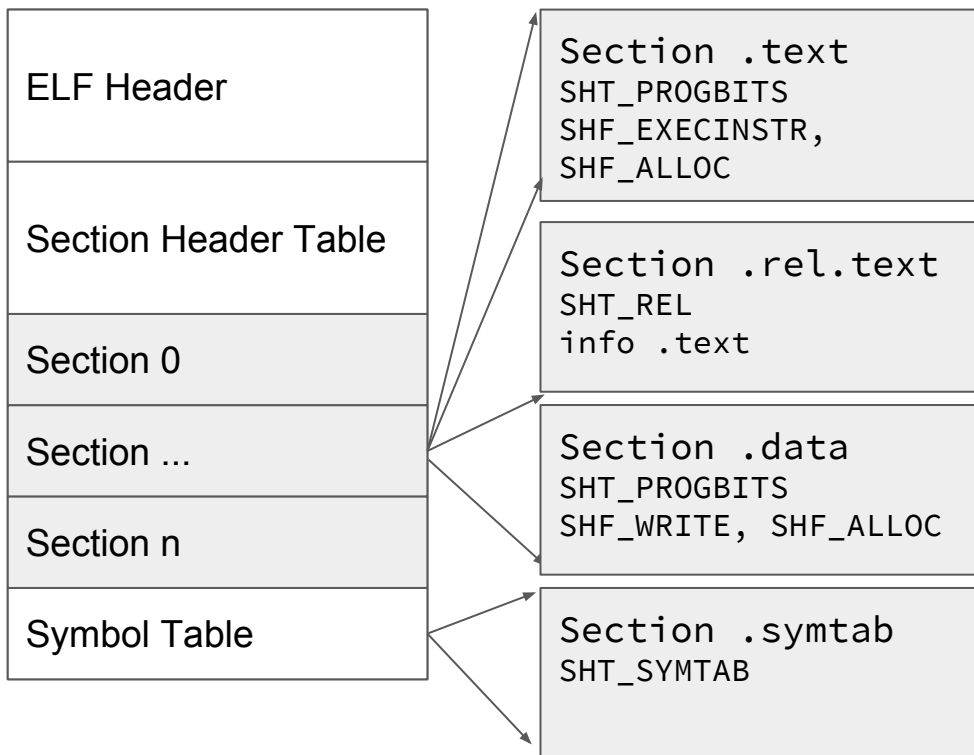


Introduction to object files

The components of an ELF File

How C and C++ programs map to
an ELF file

Components of an ELF file, Sections



- **Sections** describe ranges of bytes
- Type
 - SHT_PROGBITS
 - SHT_SYMTAB
- Flags
 - SHF_EXECINSTR
 - SHF_WRITE
 - SHF_ALLOC
- Info and Link
 - Dependencies between sections

Components of an ELF file, Symbols

```
.section .text  
.global fn1  
.type fn1, %function  
.size fn1, 0x10  
fn1:  
...
```

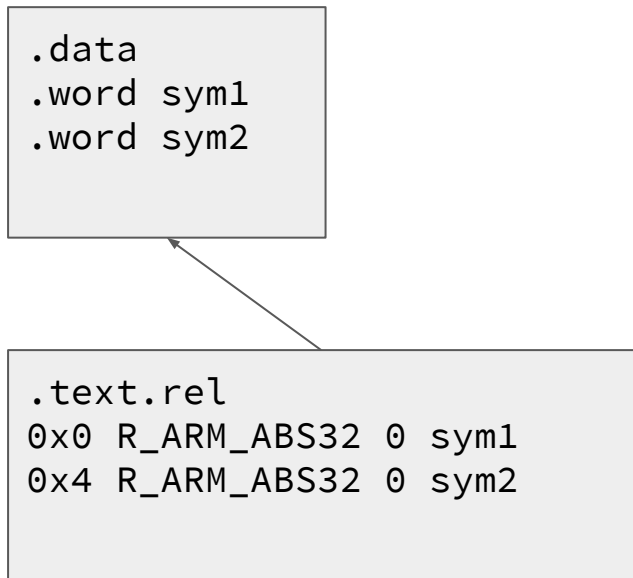
```
Name: fn1  
Value: 0x0  
Binding: STB_GLOBAL  
Visibility: STV_DEFAULT  
Type: STT_FUNC  
Size: 0x10
```

```
.local fn2  
.type fn2, %function  
.size fn2, 0x4  
fn2:  
...
```

```
Name: fn2  
Value: 0x10  
Binding: STB_LOCAL  
Visibility: STV_DEFAULT  
Type: STT_FUNC  
Size: 0x4
```

Components of an ELF file Relocations

```
.data  
.word sym1  
.word sym2
```

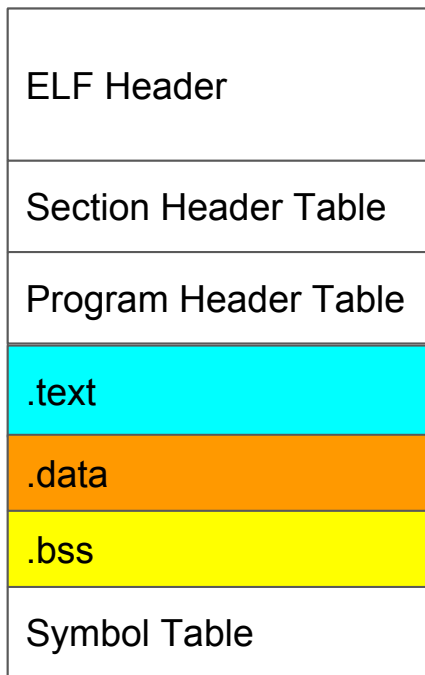


```
.text.rela  
0x0 R_ARM_ABS32 0 sym1  
0x4 R_ARM_ABS32 0 sym2
```

Relocations encode places where the linker needs to fix up locations in the output

- Relocation section is linked to a section via sh_info field
- Offset in section to apply relocation
- Relocation code to apply
- Addend
- Target symbol

Components of ELF file, Segments



PT_LOAD
PF_R, PF_X
Size in file: up to end of .text
Size in mem: .data + .bss

PT_LOAD
PF_W
Size in file: size of .data
Size in mem: size of .data and .bss

Mapping of C Program to ELF

```
#include <stdio.h>

static int rw = 10;
int zi;
static int
function2(void)
{
    return rw + zi;
}
void function1(void)
{
    rw += 1;
    printf("%d\n",
           function2());
}
```

Sections

<code>.text</code> Type: SHT_PROGBITS Flags: SHF_ALLOC, SHF_EXECINSTR
<code>rel.text</code> Type: SHT_REL
<code>.rodata.str1.1</code> Type: SHT_PROGBITS Flags: SHF_ALLOC, SHF_MERGE, SHF_STRINGS
<code>.data</code> Type: SHT_PROGBITS Flags: SHF_ALLOC, SHF_WRITE
<code>.bss</code> Type: SHT_NOBITS Flags: SHF_ALLOC, SHF_WRITE

Symbols

`rw`, STT_OBJ, STB_LOCAL, `.data`
`zi`, STT_OBJ, STB_GLOBAL, `.bss`
`function2`, STT_FUNC, STB_LOCAL, `.text`
`function1`, STT_FUNC, STB_GLOBAL, `.text`
`printf`, STT_FUNC, `0` (reference)

Relocations

`R_ARM_MOVW_ABS_NC` rw
`R_ARM_MOVT_ABS` rw
`R_ARM_MOVW_ABS_NC` zi
`R_ARM_MOVT_ABS` zi
`R_ARM_MOVW_ABS_NC` .L.str
`R_ARM_MOVT_ABS` .L.str
`R_ARM_CALL` `function2`
`R_ARM_CALL` `printf`

Mapping a C++ Program to ELF

```
namespace Bar {  
    struct Foo {  
        Foo(int x) : x_(x) {}  
        virtual int get() const {  
            return x_; }  
        virtual ~Foo();  
        int x_;  
    };  
  
    Foo::~~Foo() {}  
};  
  
int func(void) {  
    Bar::Foo f(0);  
    return f.get();  
}
```

```
// Vague Linkage  
COMDAT group section [ 1] `.group' [_ZNK3Bar3Foo3getEv]  
contains 3 sections:  
    [Index]    Name  
    [ 7]      .text._ZNK3Bar3Foo3getEv  
    [ 8]      .ARM.extab.text._ZNK3Bar3Foo3getEv  
    [ 9]      .ARM.exidx.text._ZNK3Bar3Foo3getEv  
    ...  
  
// Mangled names  
00000000 <_ZNK3Bar3Foo3getEv>:  
    0: 6840      ldr    r0, [r0, #4]  
    2: 4770      bx     lr  
    ...  
  
// TypeInfo and VTable  
39: 00000020      8 OBJECT GLOBAL DEFAULT 19 _ZTIN3Bar3FooE  
40: 00000000     20 OBJECT GLOBAL DEFAULT 19 _ZTVN3Bar3FooE  
    ...
```

Linking process

Loading content

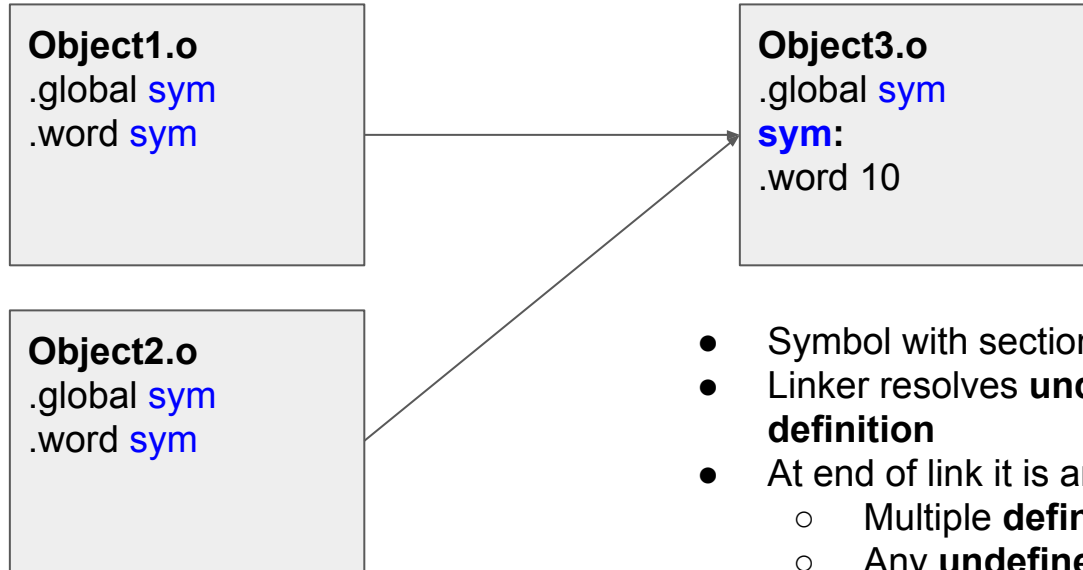
Resolving symbols

Global transformations

Section layout

Relocation

Symbol resolution



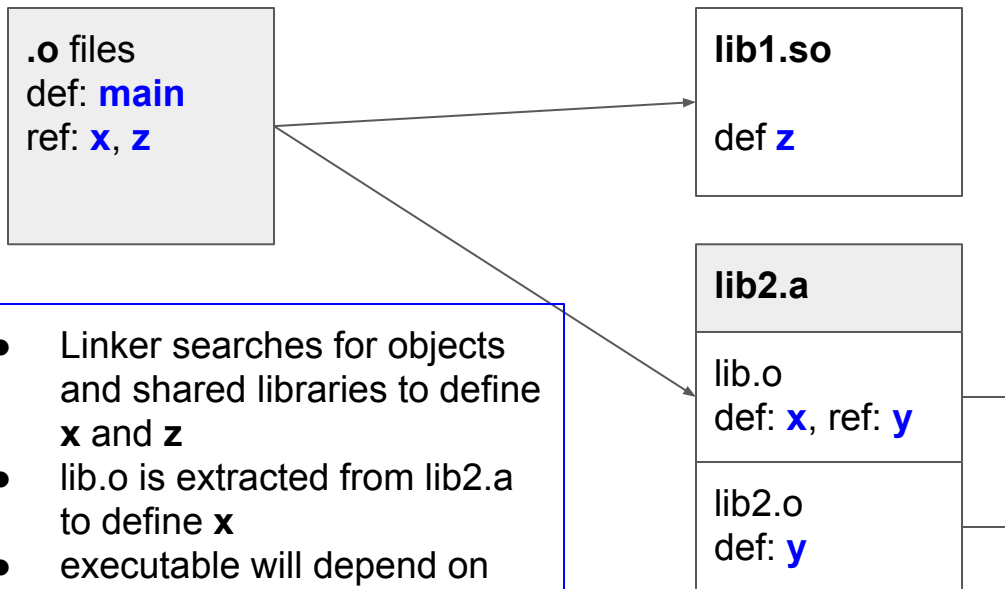
- Symbol with section index of 0 is **undefined**
- Linker resolves **undefined** symbols to a **definition**
- At end of link it is an error if:
 - Multiple **definitions** of same global symbol
 - Any **undefined** symbols still exist

Static libraries

hdr
Table of contents
hdr
Object.o
hdr
Object2.o

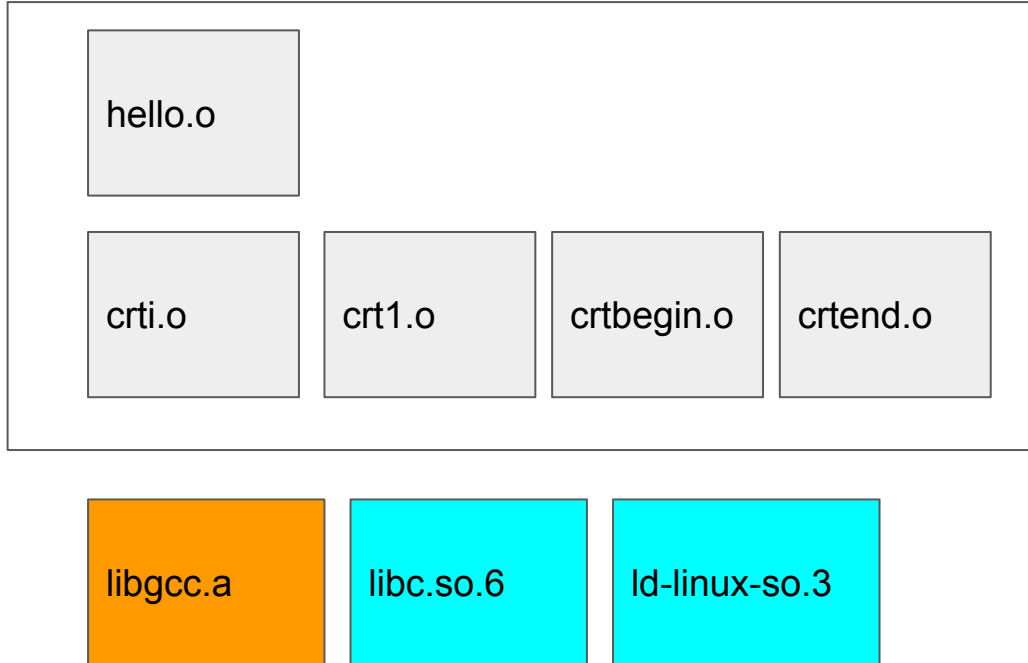
- Table of contents contains mapping of symbol **definition** to file offset of object that defines it
- Linker extracts only the objects that define **undefined** symbols
- Newly added objects may add more **undefined** symbols

Finding all the content for a link



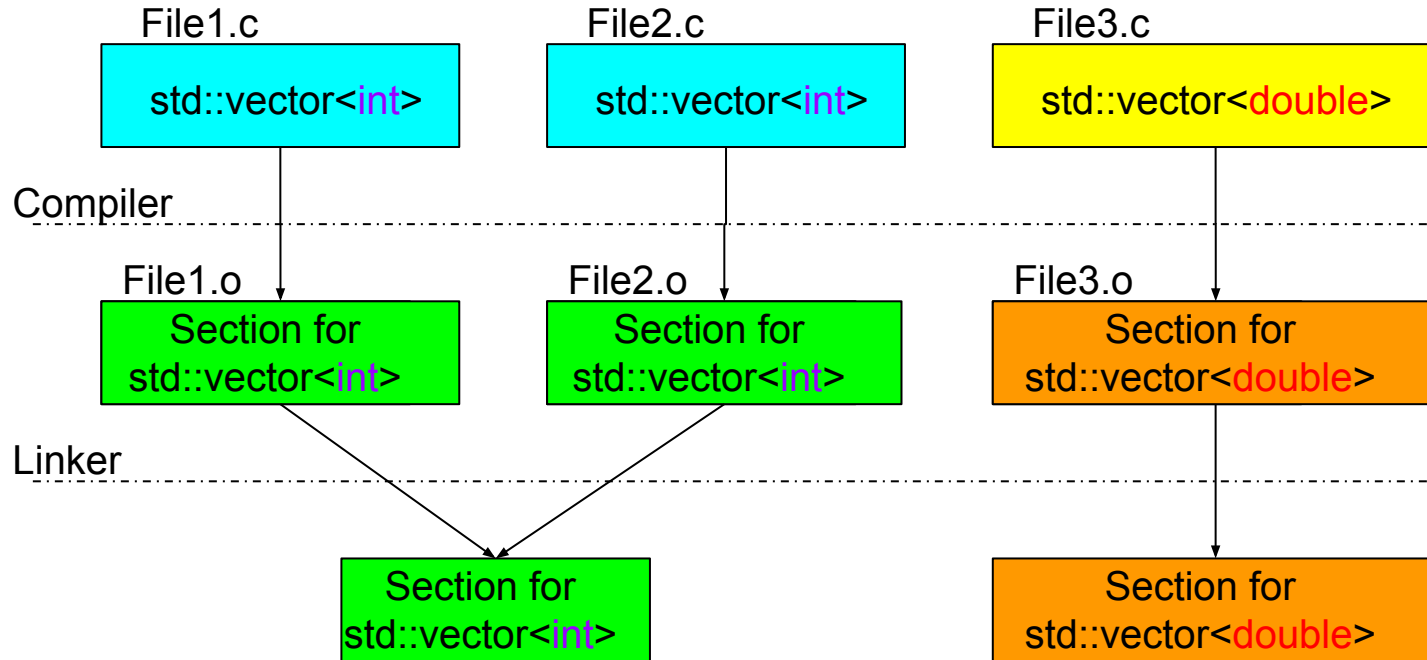
- Linker searches for objects and shared libraries to define **x** and **z**
- **lib.o** is extracted from **lib2.a** to define **x**
- executable will depend on **lib1.so** for definition of **z**
- **lib2.o** is extracted from **lib2.a** for definition of **y**, needed by **lib.o**

Content loaded for hello world



- **hello.o** produced from hello.c
- **crt*.o** added by compiler driver, contains start-up code
- **libgcc.a** is compiler support library
- **libc.so.6** contains definition of printf
- **ld-linux-so.3** is dynamic loader, needed transitively by libc.so.6

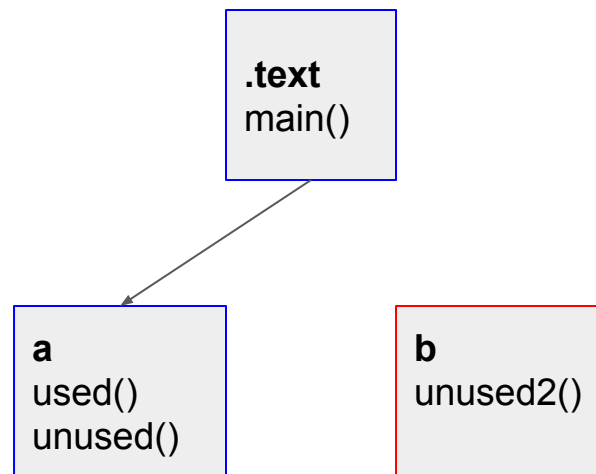
Comdat group elimination



Garbage Collection

```
void used(void)
__attribute__((section("a"))) { ... }
void unused(void)
__attribute__((section("a"))) { ... }
void unused2(void)
__attribute__((section("b"))) { ... }

int main(void) {
    used();
    return 0;
}
```



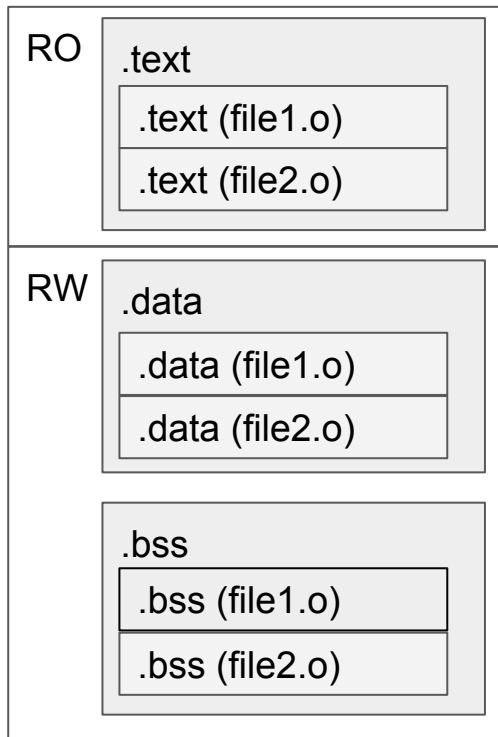
- Section **a** is reachable from entry point
- `unused()` can't be removed as linker can only remove **a**
- Section **b** is not reachable from entry point and can be removed

Layout and address allocation

0x1000

SECTIONS

```
{  
  . = 0x1000;  
  .text : { *(.text*) }  
  .data : { *(.data*) }  
  .bss  : { *(.bss*) }  
}
```

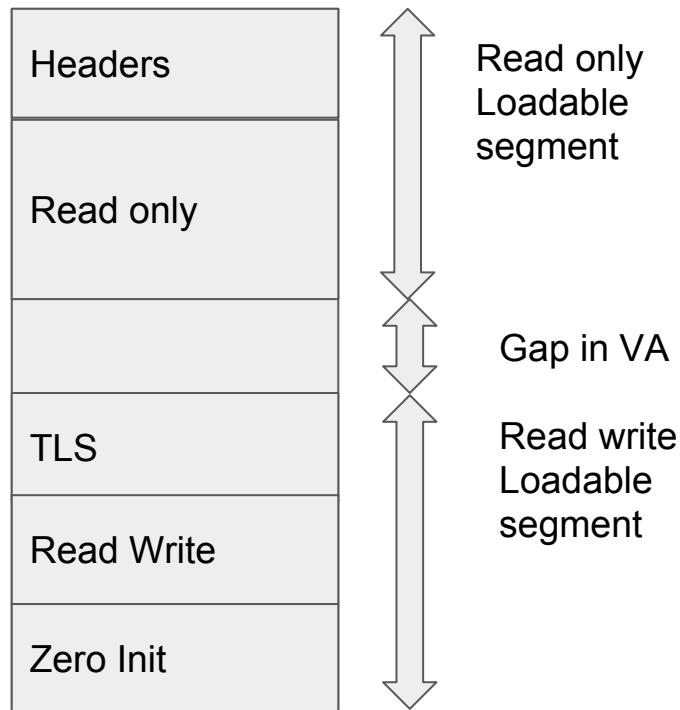


- Sections from objects
InputSections are assigned to
OutputSections
- Can be controlled by script or
by defaults
- OutputSections assigned an
address
- InputSections assigned
offsets within OutputSections
- Similar OutputSections are
described by a Program
Segment

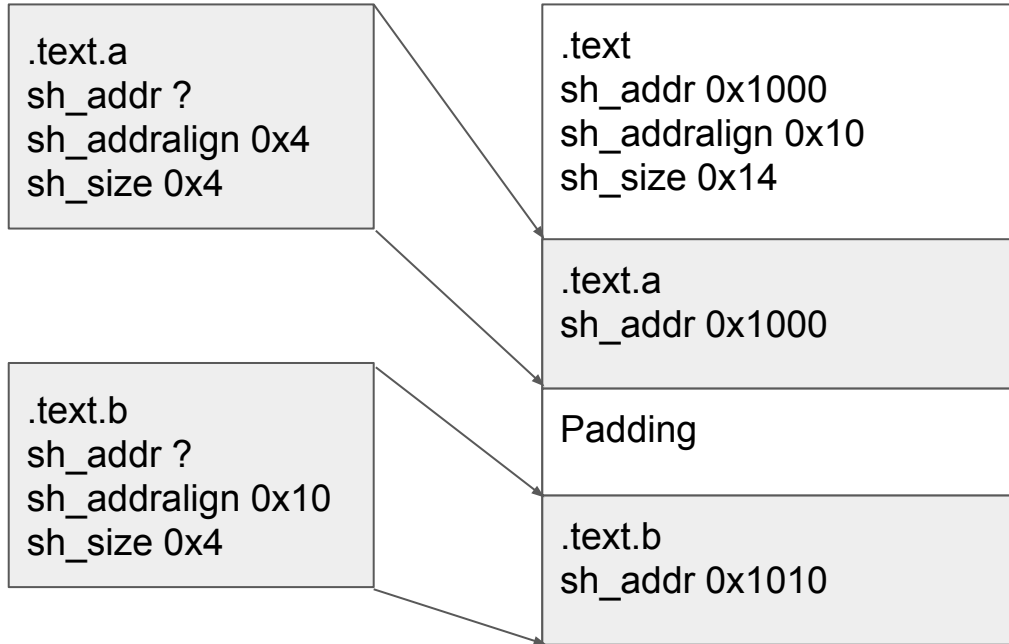
Typical layout for an executable

```
SECTIONS
{
  . = 0x10000) + SIZEOF_HEADERS;
  .interp      : { *(.interp) }
  .dynsym      : { *(.dynsym) }
  .dynstr      : { *(.dynstr) }
  ...
  .plt         : { *(.plt) }
  .text        : { *(.text*) }
  .rodata      : { *(.rodata*) }
  . = ALIGN(0x10000) + (. & (0x10000 -1));
  .tdata       : { *(.tdata*) }
  .tbss        : { *(.tbss*) }
  .dynamic     : { *(.dynamic*) }
  .got         : { *(.got*) }
  .data        : { *(.data) }
  .bss         : { *(.bss) }
}
```

0x10000

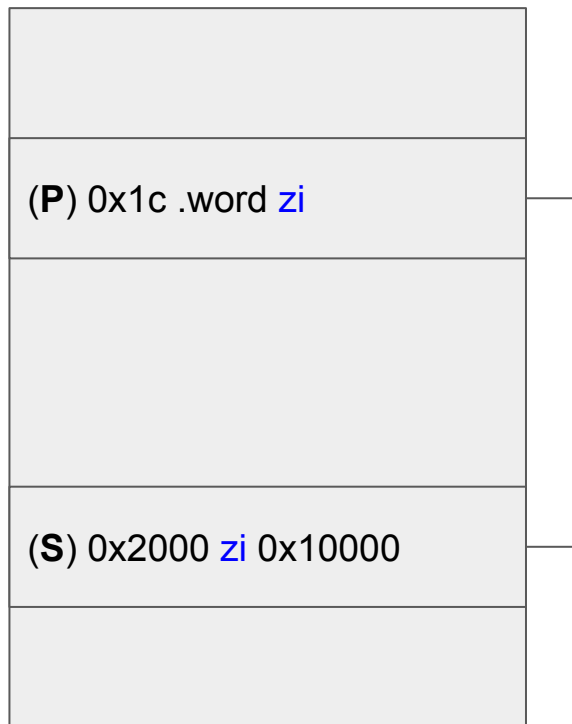


Address assignment and alignment



- OutputSection has alignment set to maximum alignment of input sections.
- Padding is inserted between InputSections to maintain alignment.

Introduction to linking: Relocation



Once final addresses of all sections are known then relocations are fixed up. In general for a relocation at address **P**

- Extract addend **A** from relocation record (RELA) or from location (REL)
- Find destination symbol address **S**
- Perform calculation
 - **S + A** for absolute
 - **S + A - P** for relative
- Write result to **P**

Relocation

```
#include <stdio.h>

static int rw = 10;
int zi;

static int function2(void)
{
    return rw + zi;
}

void function1(void)
{
    rw += 1;
    printf("%d\n",
           function2());
}
```

```
00000000 <function2>:
    0: e59f2010    ldr    r2, [pc, #16]        ; 18 <function2+0x18>
    4: e59f3010    ldr    r3, [pc, #16]        ; 1c <function2+0x1c>
    8: e5920000    ldr    r0, [r2]
    c: e5933000    ldr    r3, [r3]
   10: e0800003    add    r0, r0, r3
   14: e12fff1e    bx     lr
   18: 00000000    .word 0x00000000
   18: R_ARM_ABS32    .data
   1c: 00000000    .word 0x00000000
   1c: R_ARM_ABS32    zi

00000020 <function1>:
   20: e59f2024    ldr    r2, [pc, #36]        ; 4c <function1+0x2c>
   ...
   34: ebfffff1    bl     0 <function2>
   38: e59f1010    ldr    r1, [pc, #16]        ; 50 <function1+0x30>
   ...
   48: eaafffff    b     0 <__printf_chk>
   48: R_ARM_JUMP24    __printf_chk
   4c: 00000000    .word 0x00000000
   4c: R_ARM_ABS32    .data
   50: 00000000    .word 0x00000000
   50: R_ARM_ABS32    .rodata.str1.4
```

Relocation Example R_ARM_ABS32

```
1c: 00000000      .word   0x00000000
      1c: R_ARM_ABS32  zi
```

Relocations have the components:

- Place **P** (the offset 1c)
- Symbol **S** (the target of the relocation)
- Addend **A** (part of calculation given by compiler, 0 in this case)
- Code (R_ARM_ABS32, tells linker how to resolve relocation)

Absolute relocation such as R_ARM_ABS32 resolves as:

- **S + A**
- In this case address of `zi` + 0

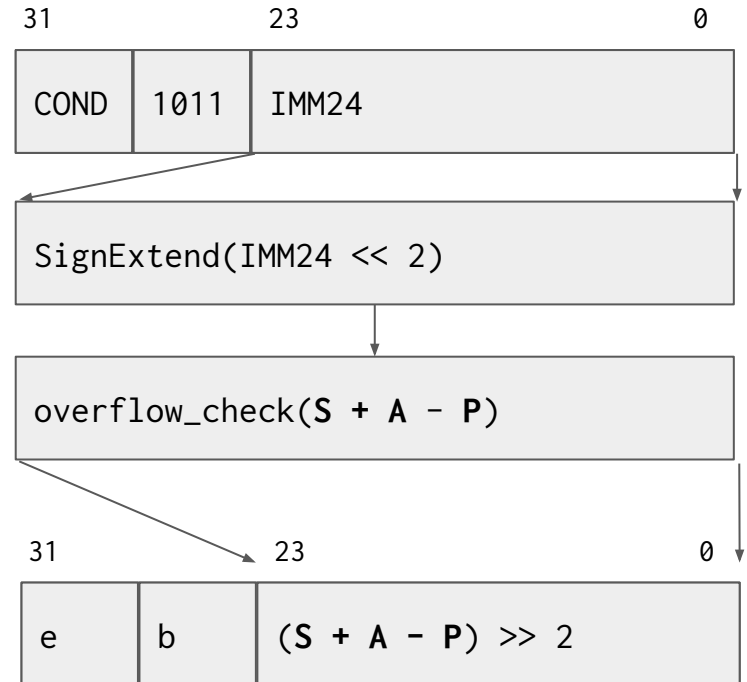
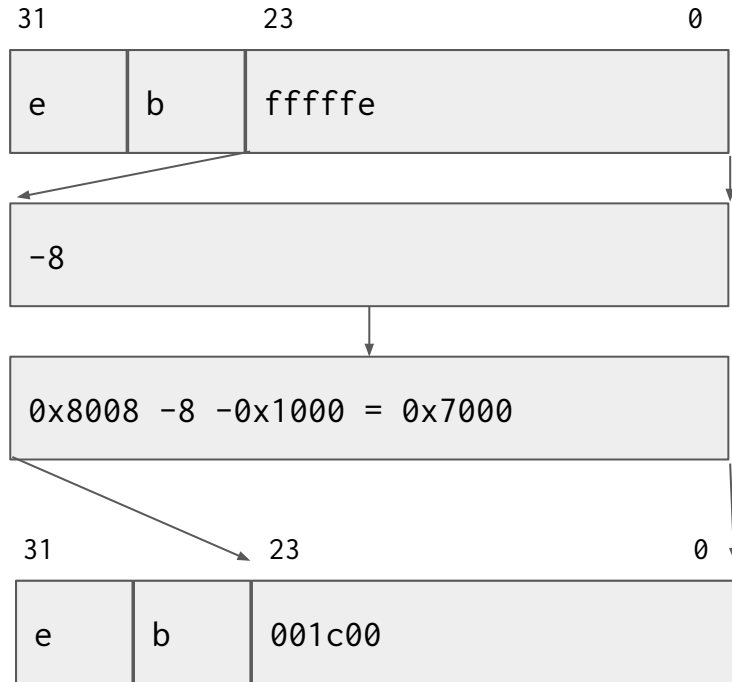
Relocation example R_ARM_PC24

```
48: eaffffffe      b    0 <__printf_chk>  
48: R_ARM_JUMP24  __printf_chk
```

R_ARM_PC24 is a relative relocation resolved as **S + A - P**

- **A** is derived from the immediate field of the instruction
 - 24 bits shifted left by 2 (ARM instructions are 4-byte aligned)
 - Resolves to -8 (ARM PC is always 2 instructions ahead)
- **P** is 0x48
- **S** is address of `__printf_chk`
- Result is `__printf_chk - 0x8 - 0x48`
- Result is range checked, right shifted by 2 and written to immediate

Relocation example R_ARM_PC24



Dynamic Linking

Position independent code

PLT and GOT generation

Symbol Visibility

Position Independent Code

```
int d;
```

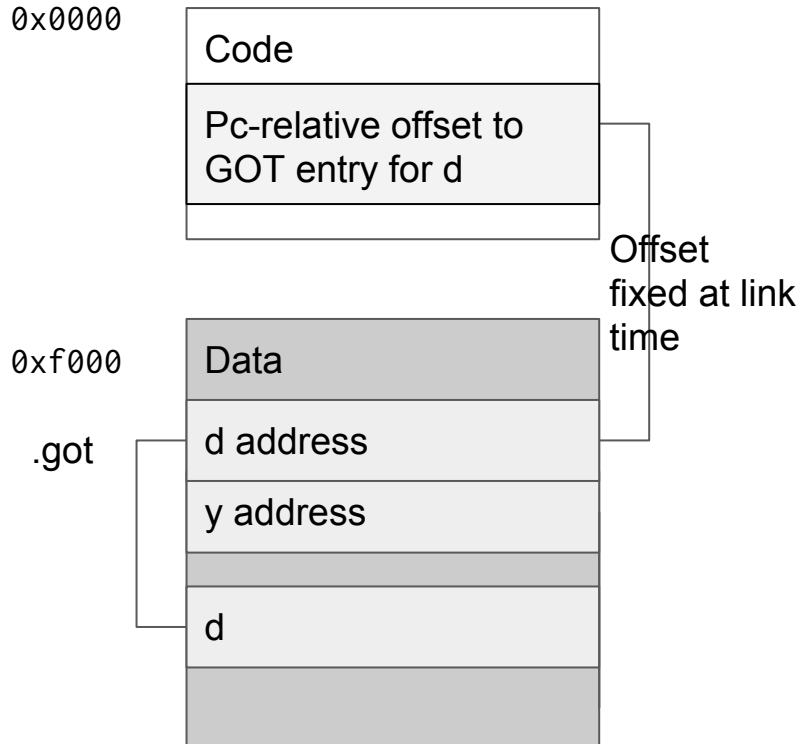
```
int function2(void) {  
    return d;  
}
```

```
int function1(void) {  
    return function1();  
}
```

```
00000000 <function>:  
    0: 4b02      ldr    r3, [pc, #8]; (c  
<function+0xc>)  
    2: 447b      add    r3, pc  
    4: 4a02      ldr    r2, [pc, #8]; (10  
<function+0x10>)  
    6: 589b      ldr    r3, [r3, r2]  
    8: 6818      ldr    r0, [r3, #0]  
    a: 4770      bx     lr  
    c: 00000006 .word 0x00000006  
                                c: R_ARM_GOTPC  
_GLOBAL_OFFSET_TABLE_  
    10: 00000000 .word 0x00000000  
                                10: R_ARM_GOT32 d
```

```
00000014 <function2>:  
    14: b508      push  {r3, lr}  
    16: f7ff fffe  bl    0 <function>  
                                16: R_ARM_THM_CALLfunction  
    1a: bd08      pop   {r3, pc}
```

Position independent code via GOT



Global Offset Table (GOT) is constructed by the linker in response to specific relocations

- Offset from code to data is known
- Code loads address of variable from GOT
- GOT filled in/relocated by dynamic linker
- Code is read-only and free of relocations

Calling functions in shared libraries

```
16: f7ff fffe    bl    0 <function>
```

```
16: R_ARM_THM_CALL    function
```

```
PLT[0]:
```

```
440: e52de004    push   {lr}           ; Special PLT entry to enter dynamic loader
444: e59fe004    ldr    lr, [pc, #4]   ; have to use lr register as ip is in use
448: e08fe00e    add    lr, pc, lr     ; dynamic loader restores lr
44c: e5bef008    ldr    pc, [lr, #8]!  ; jump to dynamic loader to find symbol
450: 00010bb0    .word  0x00010bb0
...
```

```
PLT[function]:
```

```
46c: e28fc600    add    ip, pc, #0, 12
470: e28cca10    add    ip, ip, #16, 20
474: e5bcfba0    ldr    pc, [ip, #2976]! ; jump to .plt.got[function]
...
```

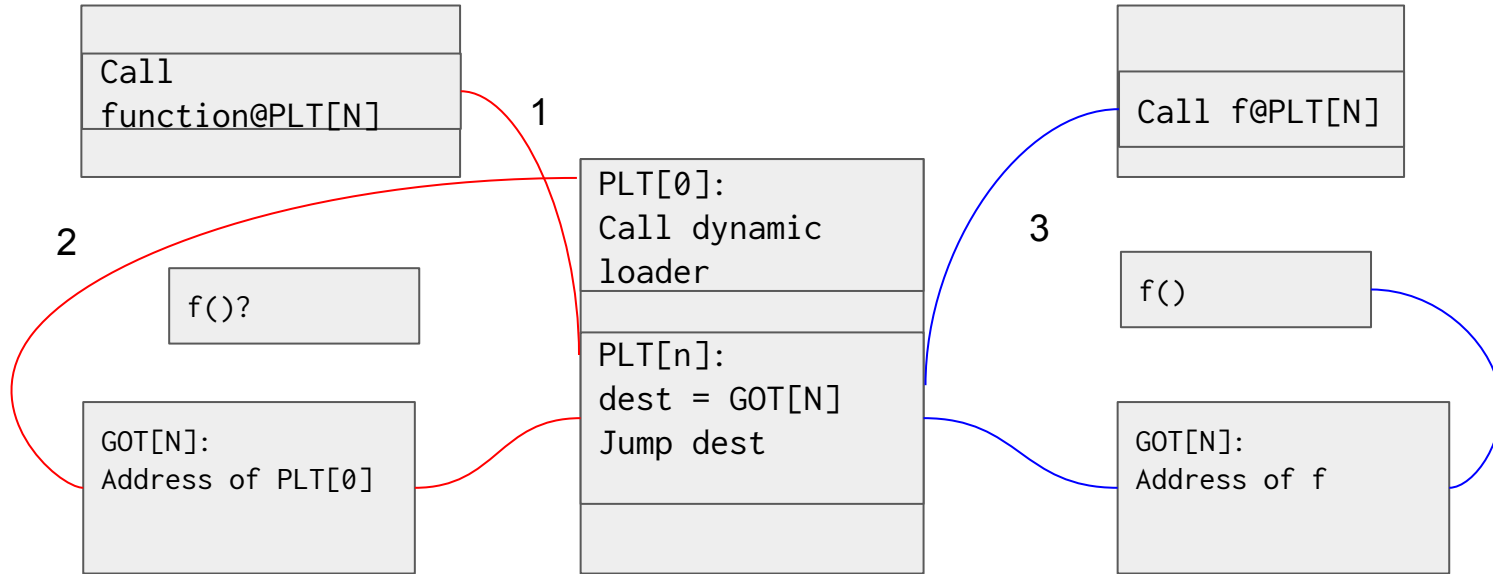
```
.plt.got[0]
```

```
.word "entry point to dynamic loader"
```

```
.plt.got[function]
```

```
.word "address of function", initially points to PLT[0]
```

Calling a function via PLT



Lazy binding, 1st call

Subsequent calls

Copy Relocation and preemption

```
// main
extern int x;
extern void function();
extern void function2();

int main(void) {
    x = 0;
    function();
    function2();
    return x;
}
```

All uses of x must use the same definition.

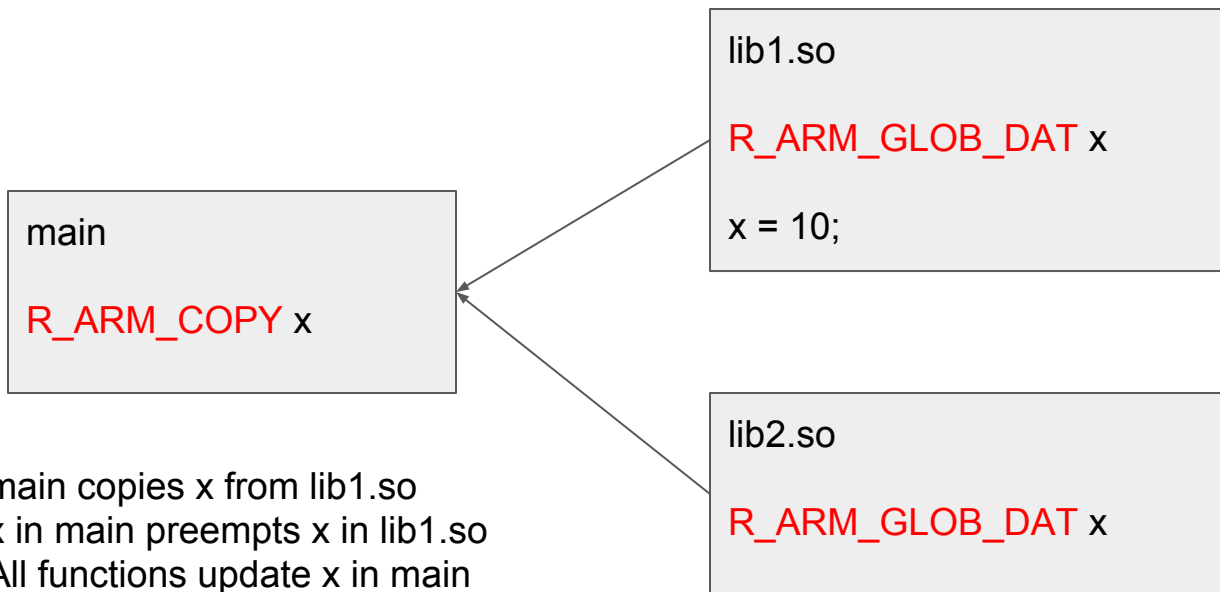
```
// lib1.so
int x = 10;

void function(void) {
    x += 1;
}

// lib2.so
extern int x;

void function2(void) {
    x += 2;
}
```

Copy Relocation and preemption



Symbol visibility

- Symbol preemption useful to make semantics of using shared libraries close to static libraries
 - Has some performance penalty as compiler and linker have to assume symbol can be preempted at run time.
- Can be influenced by symbol visibility

Symbol Visibility	Visible outside component	Preemptible
STV_DEFAULT	Yes	Yes
STV_PROTECTED	Yes	No
STV_HIDDEN	No	No
STV_INTERNAL	No	No

Symbol visibility control

- Compile time
 - `int` function(`void`) `__attribute__((visibility("hidden")))`;
 - `-fvisibility=[default|internal|hidden|protected]`
- Link time with a version script `-Wl,--version-script=sym.ver`

```
VER1 {  
    global: index; // index has the global version with default visibility  
    local: *;      // everything else has the local version with hidden  
};
```

- Some caveats exist with C++ and non default visibility, especially classes:
 - Thrown across component boundaries
 - With static data members

Advanced topics

Thread local storage

Link time optimisation

Thread local storage

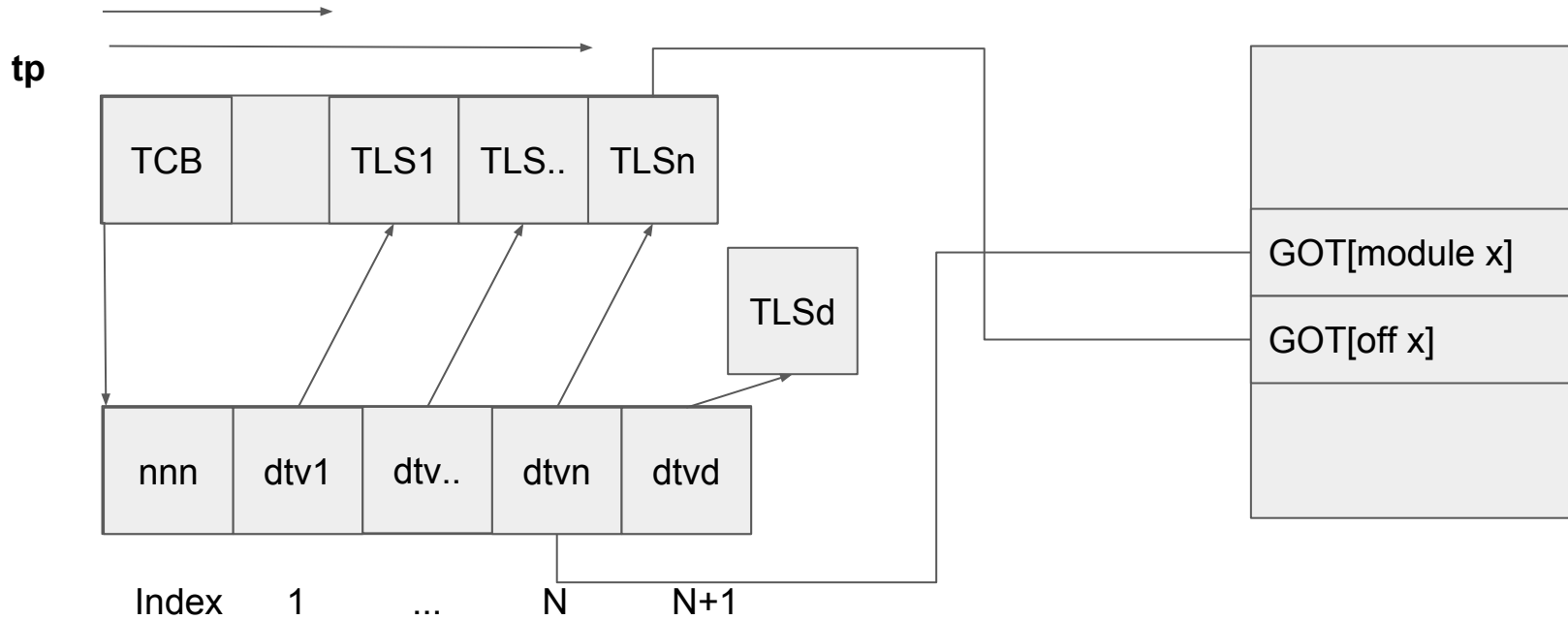
```
__thread int x
__attribute__((tls_model("global-dynamic")))
= 10;

int function(void)
{
    return x;
}
```

```
00000000 <function>:
0: 4803      ldr    r0, [pc, #12]      ; (10
<function+0x10>)
2: b508      push  {r3, lr}
4: 4478      add   r0, pc
6: f7ff fffe bl    0 <__tls_get_addr>
6: R_ARM_THM_CALL

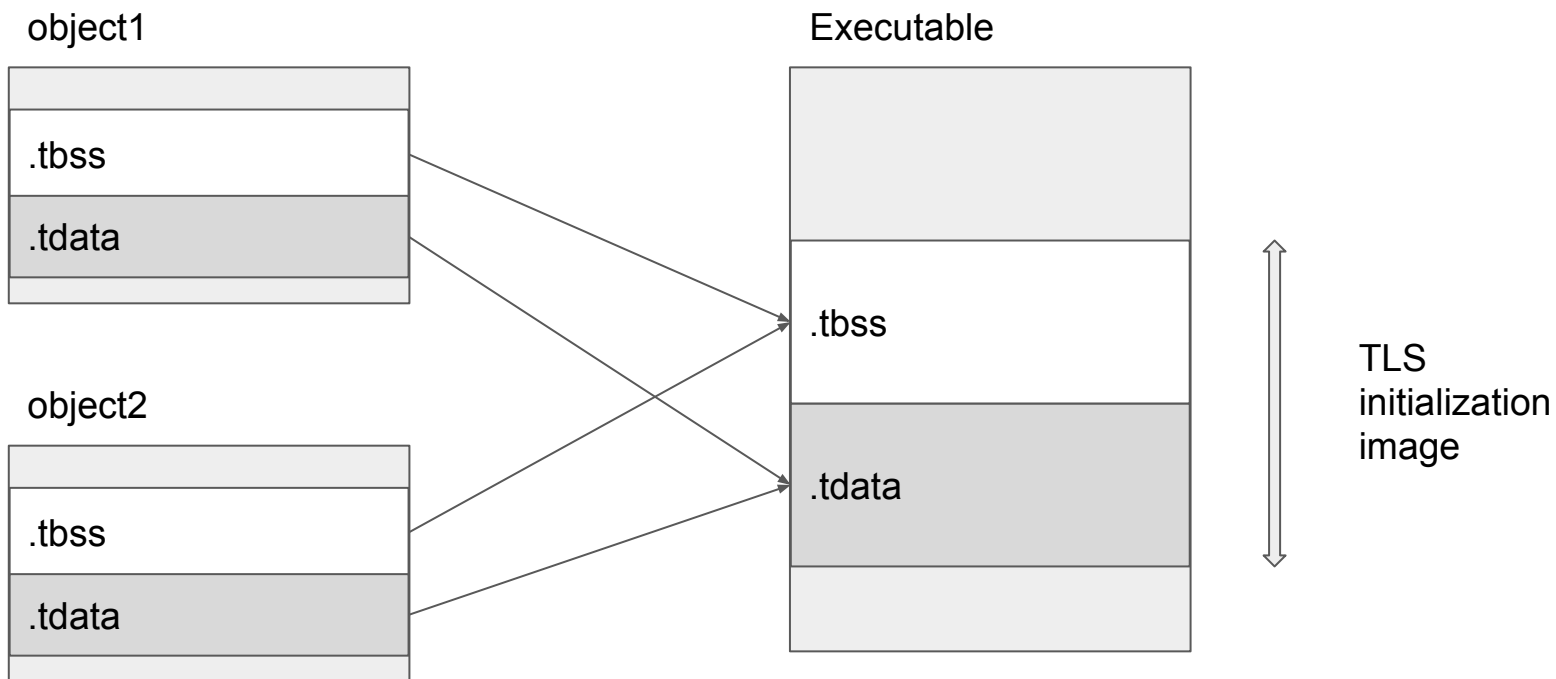
__tls_get_addr
a: 6800      ldr    r0, [r0, #0]
c: bd08      pop   {r3, pc}
e: bf00      nop
10: 00000008 .word 0x00000008
10: R_ARM_TLS_GD32      .LANCHOR0
```


Thread local storage (global dynamic)



$(\mathbf{tp}[0])[\mathbf{GOT}[\text{module } x]] + \mathbf{GOT}[\text{off } x]$

Thread local storage



Link time optimisation

```
// a.c
extern void foo4(void);

static signed int i = 0;

void foo2(void) {
    i = -1;
}

static int foo3() {
    foo4();
    return 10;
}

int foo1(void) {
    int data = 0;
    if (i < 0)
        data = foo3();
    data = data + 42;
    return data;
}
```

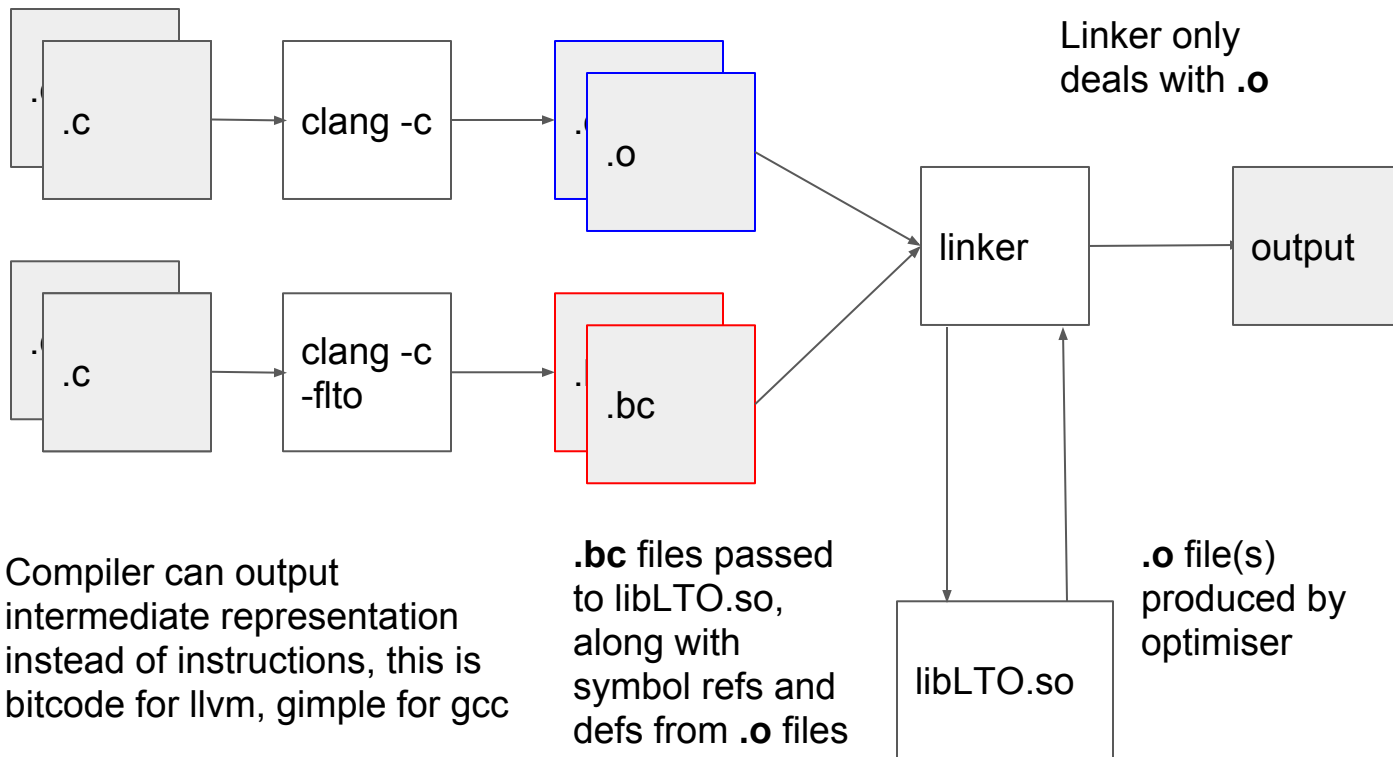
```
// b.c
extern int foo4(void);

void foo4(void) {
    printf("Hi\n");
}

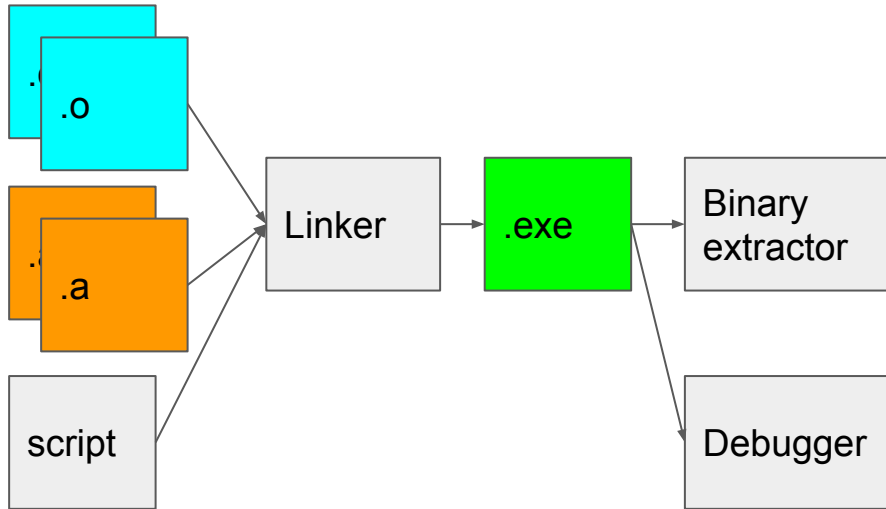
int main() {
    return foo1();
}
```

- Example from llvm.org/docs/LinkTimeOptimization.html
- foo4(), foo3() can be removed when compiler has visibility of whole program
 - foo2() is never called so i is always 0
- main can return 42

LTO implementation (llvm)



Considerations for embedded systems



- Cross linker, runs on Windows or Linux “host”, inputs and outputs are for the “target” device
 - Different architecture?
 - Different OS?
 - Different endianness?
- RTOS that runs on device is frequently a static library
- Memory layout of program controlled by a linker script
 - Maybe non-contiguous
- ELF file needs to be converted to another format, frequently binary or hex to run
- Debug information present in ELF file

Concluding thoughts

Invoking the linker

The linker map file

References and useful tools

Invoking the system linker on linux

- Can be invoked directly, but in general it is more manageable to invoke via the compiler driver
 - Ensures that the C-library startup objects and libraries are added
- The ld program is often a symlink to either ld.bfd (GNU linker) or ld.gold
 - Compiler command line option `-fuse-ld=<linker>` can be used with `bfd` or `gold`
 - Clang also supports `lld`
- Use `-v` to see the invocation that gcc/clang passes to ld

Example extract of linker invocation

```
ld --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu
--as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro -o
hello /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o
-L/usr/lib/gcc/x86_64-linux-gnu/5
-L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib
-L/lib/x86_64-linux-gnu -L/lib/./lib -L/usr/lib/x86_64-linux-gnu
-L/usr/lib/./lib -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../
/tmp/ccDXXyvf.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc
--as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/x86_64-linux-gnu/5/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crtn.o
```


Linker Map Files -Wl,--map=<file.txt>

```
.text          0x000000000004004e0      0x182
.text.unlikely
               0x000000000004004e0      0x0
/usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o
.text.unlikely
               0x000000000004004e0      0x0
/usr/lib/x86_64-linux-gnu/libc_nonshared.a(elf-init.oS)
.text          0x000000000004004e0      0x2a
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o
               0x000000000004004e0      _start
.text          0x0000000000040050a      0x0
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crti.o
** fill      0x0000000000040050a      0x6
.text          0x00000000000400510      0xc6
/usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o
.text          0x000000000004005d6      0x15 /tmp/cc2UsNK8.o
               0x000000000004005d6      main
```

References

- Ian Lance Taylor's blog
 - <http://www.airs.com/blog/archives/38>
- Linker and Loaders book
 - <https://www.iecc.com/linker/>
- ELF Standard
 - <http://www.sco.com/developers/gabi/>
- IA64 C++ ABI
 - <https://itanium-cxx-abi.github.io/cxx-abi/abi.html>
- How to write shared libraries
 - <https://www.akkadia.org/drepper/dsohowto.pdf>

Useful tools

- readelf
 - **--sections, --symbols, --relocs**
- objdump
 - disassembly **-d**, relocations **-r**
- Compiler options
 - Passing options: **-Wl,option -Xlinker option**
 - Sub-process invocation of linker **-v**
- Selecting a linker from gcc/clang
 - **-fuse-ld=bfd, -fuse-ld=gold** selects ld.bfd, ld.gold
- GNU linker map files
 - **-Wl,-Map=map.txt**
- GNU linker print default script to stdout
 - **ld --verbose**