计算机系统基础10

Linking

Today

- **Linking**
  链接
  - Motivation
  - What it does
  - How it works
  - Dynamic linking
- **Position-Independent Code（PIC）**
  位置无关代码

Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

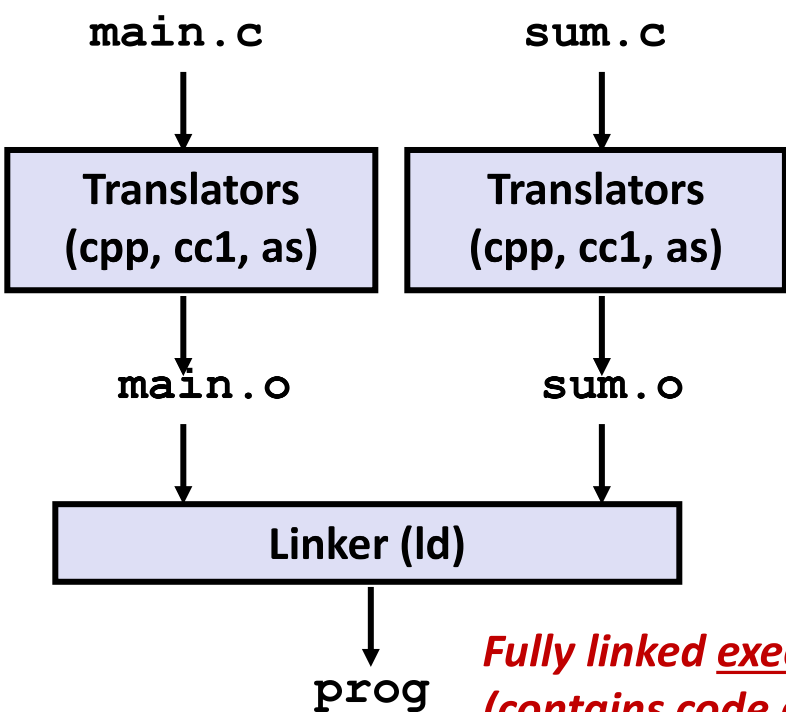
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Source files

Separately compiled <u>relocatable</u> object files

Fully linked <u>executable</u> object file (contains code and data for all functions defined in `main.c` and `sum.c`)

Why Linkers?

- **Reason 1: Modularity**

  - Program can be written as a collection of smaller source files, rather than one monolithic mass.

  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

Why Linkers? (cont)

- **Reason 2: Efficiency**
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
    - Can compile multiple files concurrently.
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - **Option 1: *Static Linking***
      - Executable files and running memory images contain only the library code they actually use
    - **Option 2: *Dynamic linking***
      - Executable files contain no library code
      - During execution, single copy of library code can be shared across all executing processes

What Do Linkers Do?

- **Step 1: Symbol resolution**

  解析

  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol swap */`
    - `int *xp = &x;      /* define symbol xp, reference x */`

  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol.

  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

Symbols in Example C Program

Definitions

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Reference

What Do Linkers Do? (cont)

- **Step 2: Relocation**
  重定位
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Let's look at these two steps in more detail....

Three Kinds of Object Files (Modules)

- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (`.so` file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

Executable and Linkable Format (ELF)

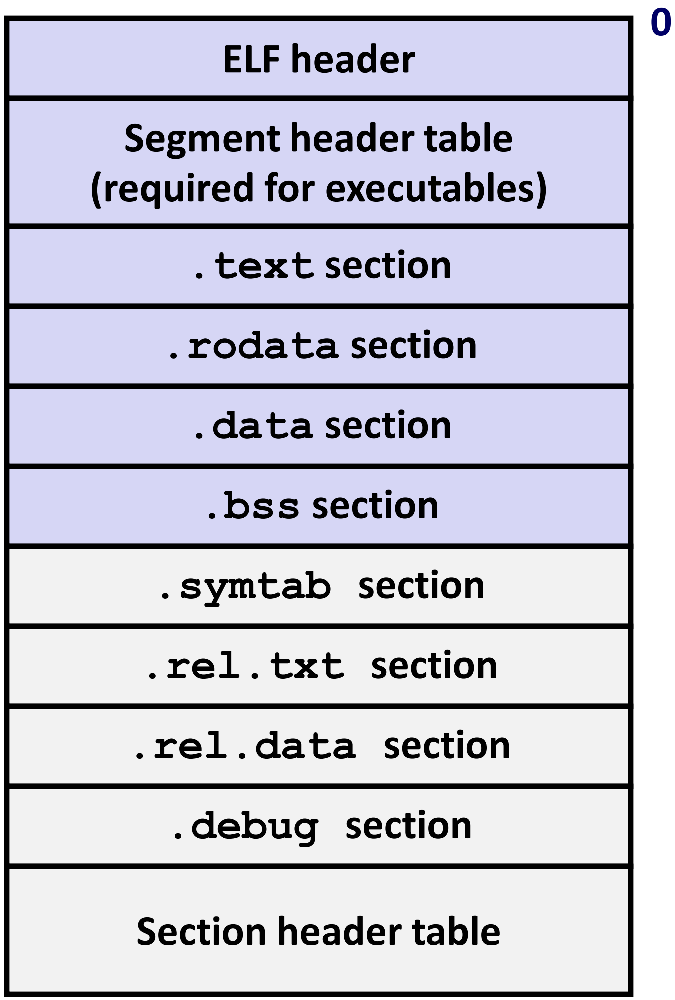
- **Standard binary format for object files**

- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- **Generic name: ELF binaries**

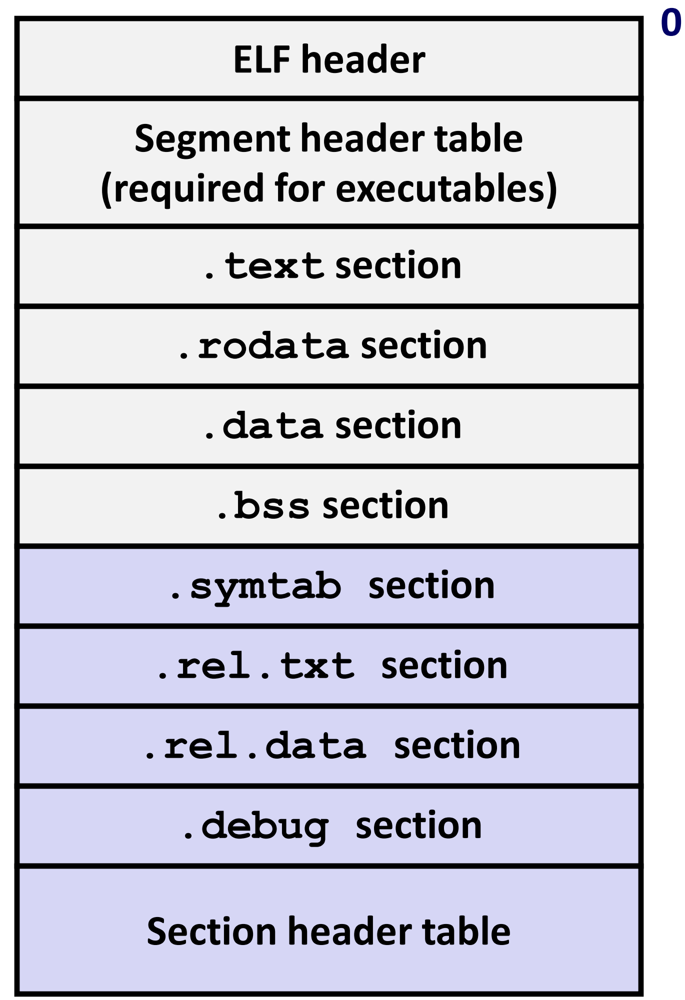
ELF Object File Format

- **ELF header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, string constants, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - “Block Storage Start”
  - “Better Save Space”
  - Has section header but occupies no space



ELF Object File Format (cont.)

- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the relocatable ~~executable~~
  - Instructions for modifying.
- **`.rel.data` section**
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (**`gcc -g`**)
- **Section header table**
  - Offsets and sizes of each section



Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-**`static`** C functions and non-**`static`** global variables.
- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module.
- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and global variables defined with the **`static`** attribute.
  - **Local linker symbols are *not* local program variables**

Step 1: Symbol Resolution

Referencing a global…

…that's defined here

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

Symbol Identification

Which of the following names will be in the symbol table of `symbols.o`?

symbols.c:

```
int incr = 1;

static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

Names:

- `time`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Local Symbols

- **Local non-static C variables vs. local static C variables**
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss,` or `.data`

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

static-local.c

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

How Linker Resolves Duplicate Symbol Definitions

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals
    - Or ones declared with specifier **`extern`**

strong → p1.c

```
int foo=5;

p1() {
}
```

strong →

p2.c ← weak

```
int foo;

p2() {
}
```

← strong

Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc –fno-common`
- **Puzzles on the next slide**

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**`p1`**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **`x`** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **`x`** in **`p2`** might overwrite **`y`**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **`x`** in **`p2`** will overwrite **`y`**! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **`x`** will refer to the same initialized variable.

Important: Linker does not do type checking.

Type Mismatch Example

```c
long int x;  /* Weak symbol */

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

mismatch-main.c

```c
/* Global strong symbol */
double x = 3.14;
```

mismatch-variable.c

- Compiles without any errors or warnings
- What gets printed?

```
-bash-4.2$ ./mismatch
4614253070214989087
```

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if not defined in some file

Use of `extern` in .h Files (#1)

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include "global.h"

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Use of .h Files (#2)

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
  extern int g;
  static int init = 0;
#else
  extern int g;
  static int init = 0;
#endif
```

c2.c

```
#define INITIALIZE
#include <stdio.h>
#include "global.h"

int main(int argc, char** argv) {
  if (init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

```
int g = 23;
static int init = 1;
```

Linking Example

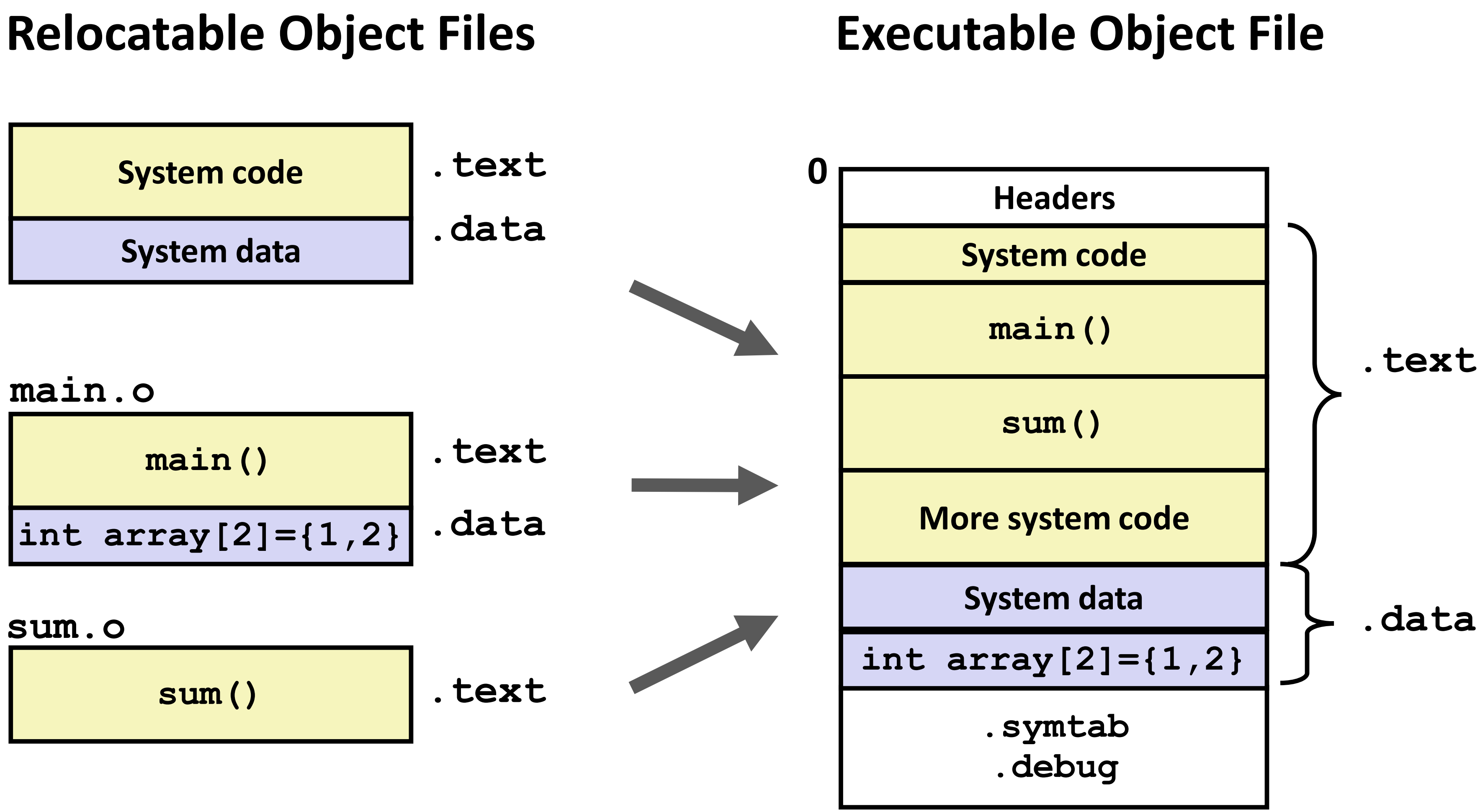
```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Step 2: Relocation



Relocation Entries

```c
int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08                 sub     $0x8,%rsp
   4:   be 02 00 00 00              mov     $0x2,%esi
   9:   bf 00 00 00 00              mov     $0x0,%edi        # %edi = &array
                        a: R_X86_64_32 array                 # Relocation entry

   e:   e8 00 00 00 00              callq   13 <main+0x13>  # sum()
                        f: R_X86_64_PC32 sum-0x4             # Relocation entry
  13:   48 83 c4 08                 add     $0x8,%rsp
  17:   c3                          retq
```

main.o

Source: `objdump -r -d main.o`

Relocated .text section

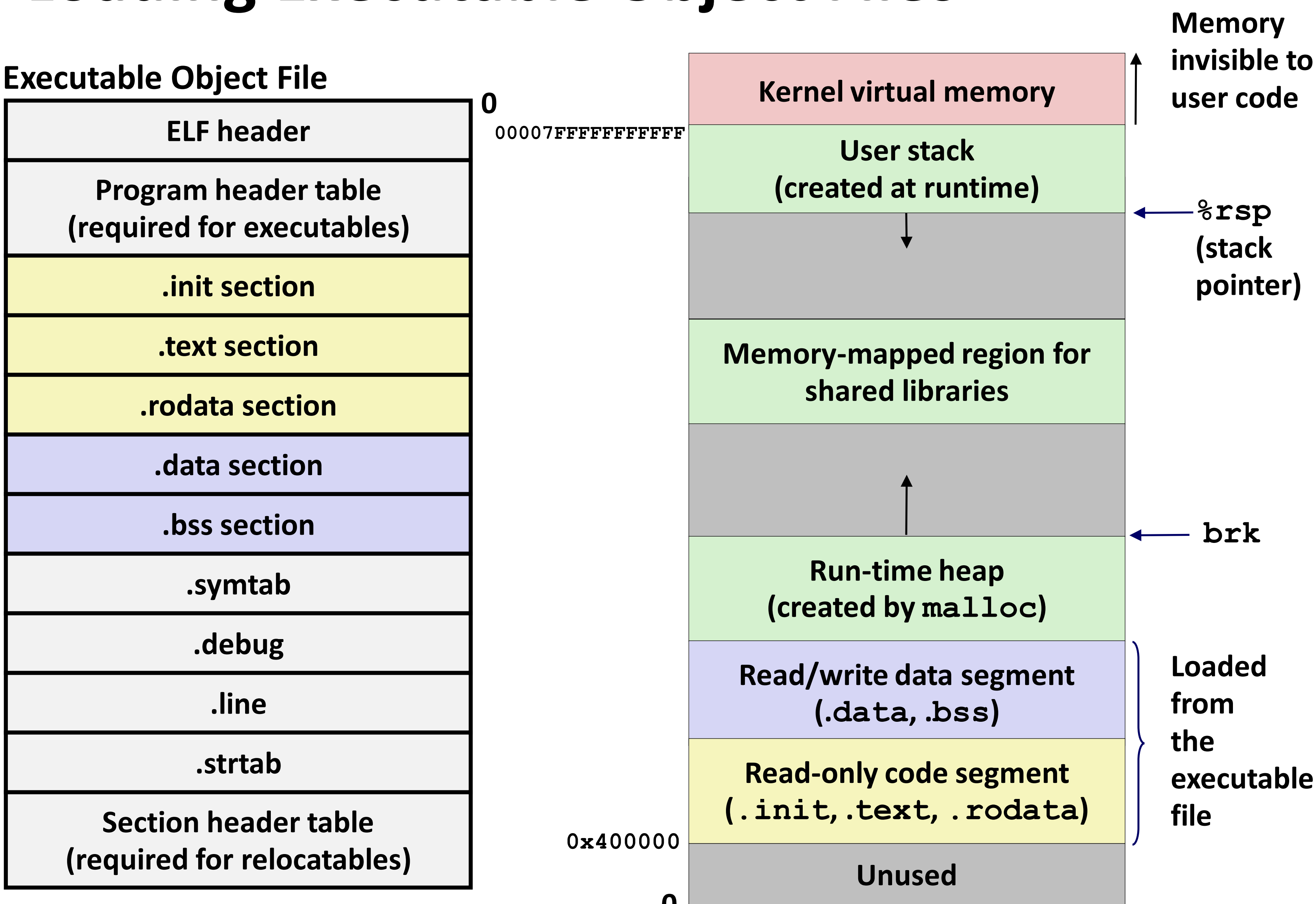
```
00000000004004d0 <main>:
  4004d0:        48 83 ec 08          sub    $0x8,%rsp
  4004d4:        be 02 00 00 00       mov    $0x2,%esi
  4004d9:        bf 18 10 60 00       mov    $0x601018,%edi  # %edi = &array
  4004de:        e8 05 00 00 00       callq  4004e8 <sum>    # sum()
  4004e3:        48 83 c4 08          add    $0x8,%rsp
  4004e7:        c3                   retq

00000000004004e8 <sum>:
  4004e8:        b8 00 00 00 00               mov    $0x0,%eax
  4004ed:        ba 00 00 00 00               mov    $0x0,%edx
  4004f2:        eb 09                        jmp    4004fd <sum+0x15>
  4004f4:        48 63 ca                     movslq %edx,%rcx
  4004f7:        03 04 8f                     add    (%rdi,%rcx,4),%eax
  4004fa:        83 c2 01                     add    $0x1,%edx
  4004fd:        39 f2                        cmp    %esi,%edx
  4004ff:        7c f3                        jl     4004f4 <sum+0xc>
  400501:        f3 c3                        repz retq
```

`callq` instruction uses PC-relative addressing for sum():

`0x4004e8` = `0x4004e3` + `0x5`

Source: `objdump -d prog`

Loading Executable Object Files



Packaging Commonly Used Functions

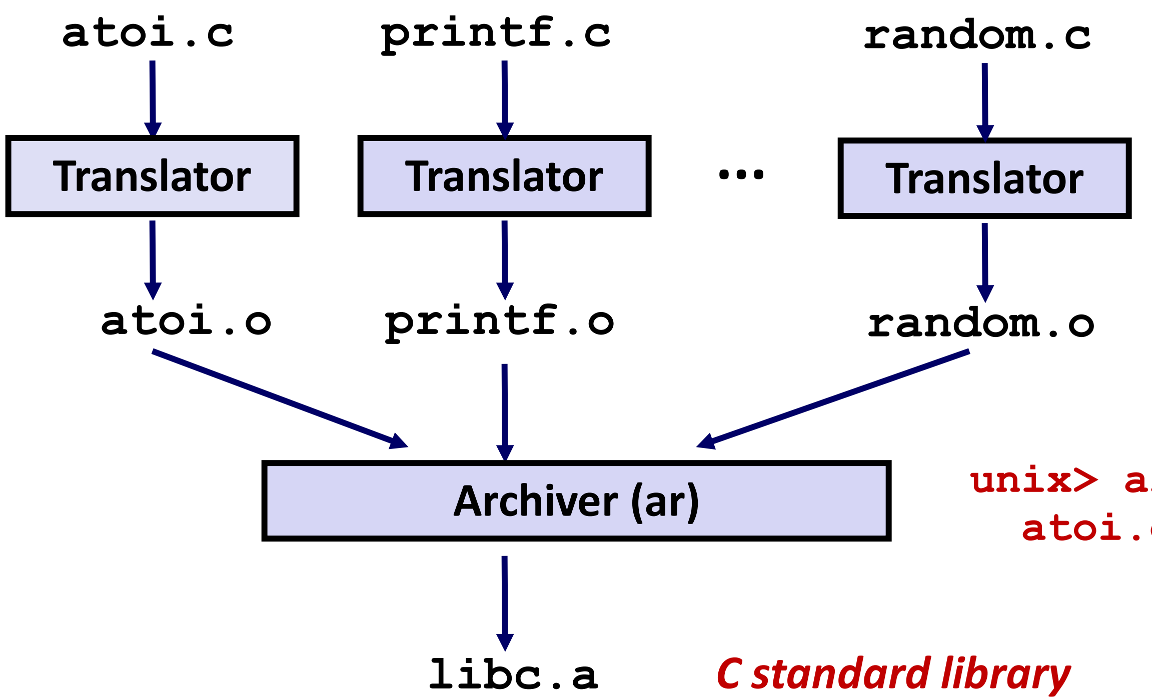
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-fashioned Solution: Static Libraries

- **Static libraries (.a archive files)**
  - Concatenate related relocatable object files into a single file with an index 连接 (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

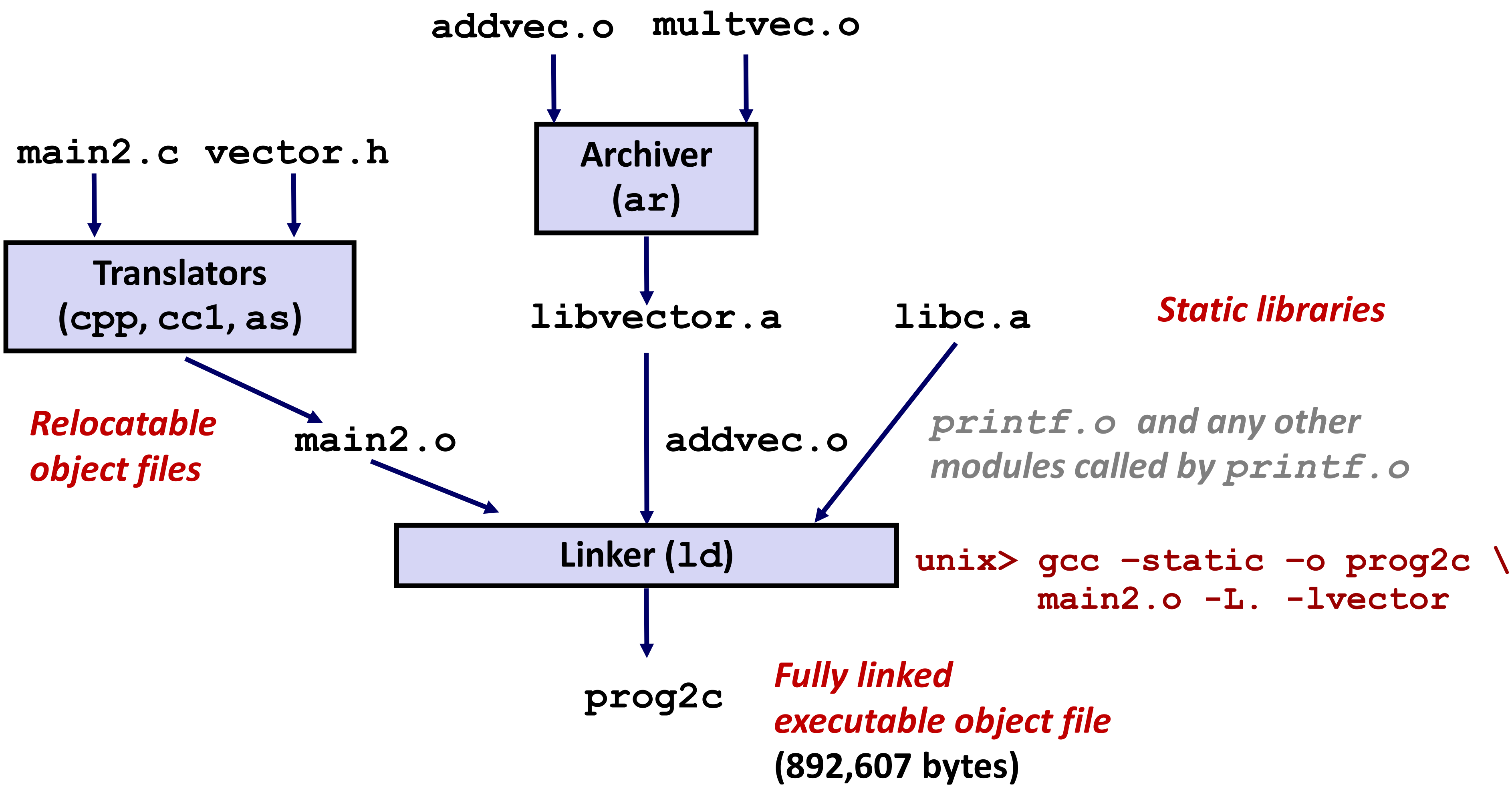
```c
int addcnt = 0;
void addvec(int *x, int *y,
            int *z, int n) {
    int i;
    addcnt++;
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```c
int multcnt = 0;
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;
    multcnt++;
    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries



"c" for "compile-time"

Using Static Libraries

- **Linker's algorithm for resolving external references:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.

打桩
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual memory

What dynamic libraries are required?

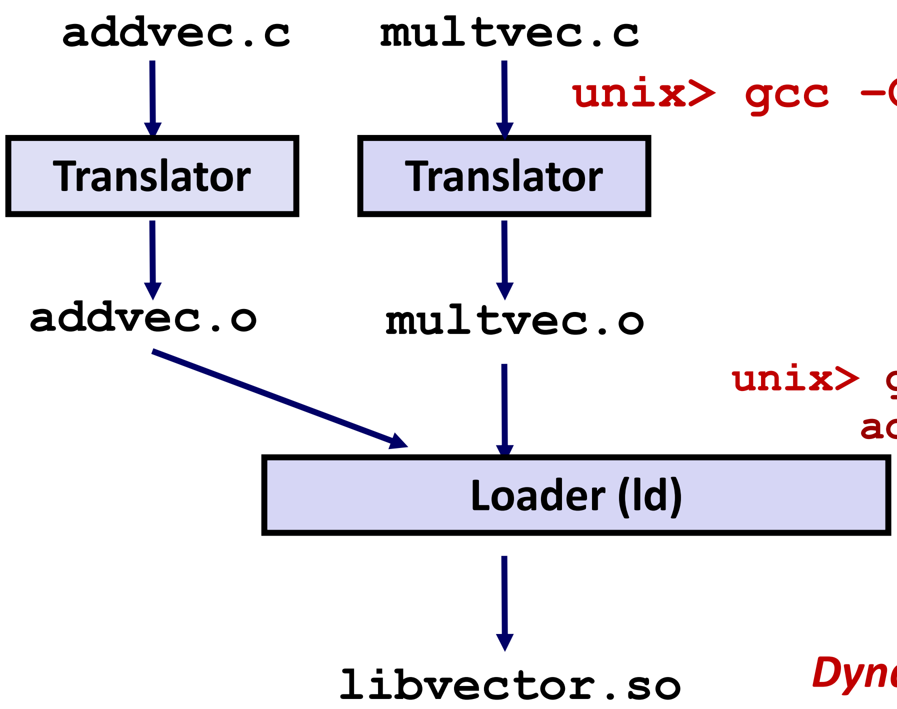
- **.interp section**
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **.dynamic section**
  - Specifies the names, etc of the dynamic libraries to use
  - Follow an example of **`prog`**

```
(NEEDED)                     Shared library: [libm.so.6]
```

- **Where are the libraries found?**
  - Use "**`ldd`**" to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

Dynamic Library Example

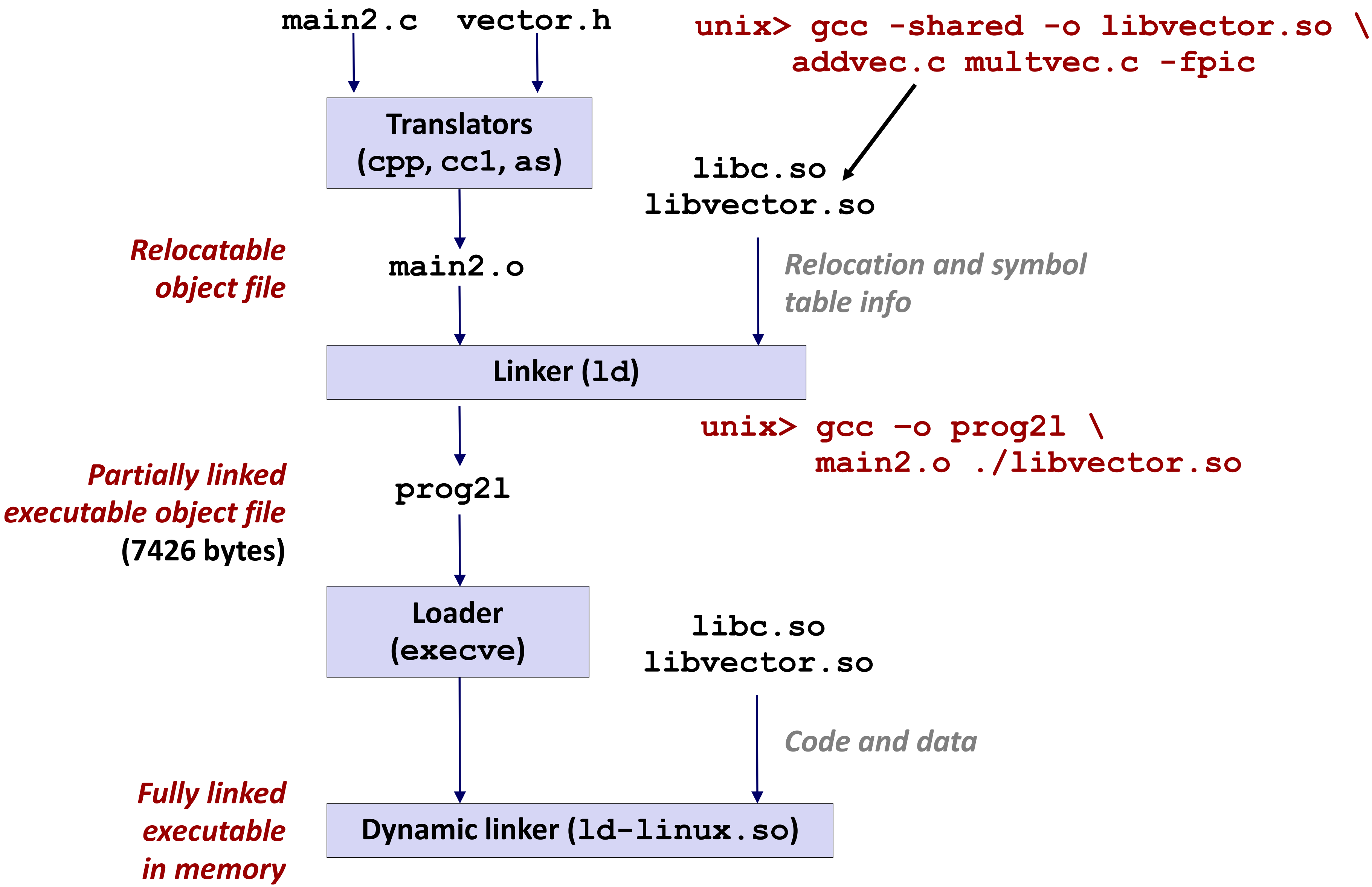



```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
   addvec.o multvec.o
```

Dynamic vector library

Dynamic Linking at Load-time



Dynamic Linking at Run-time

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

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

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
  . . .
```

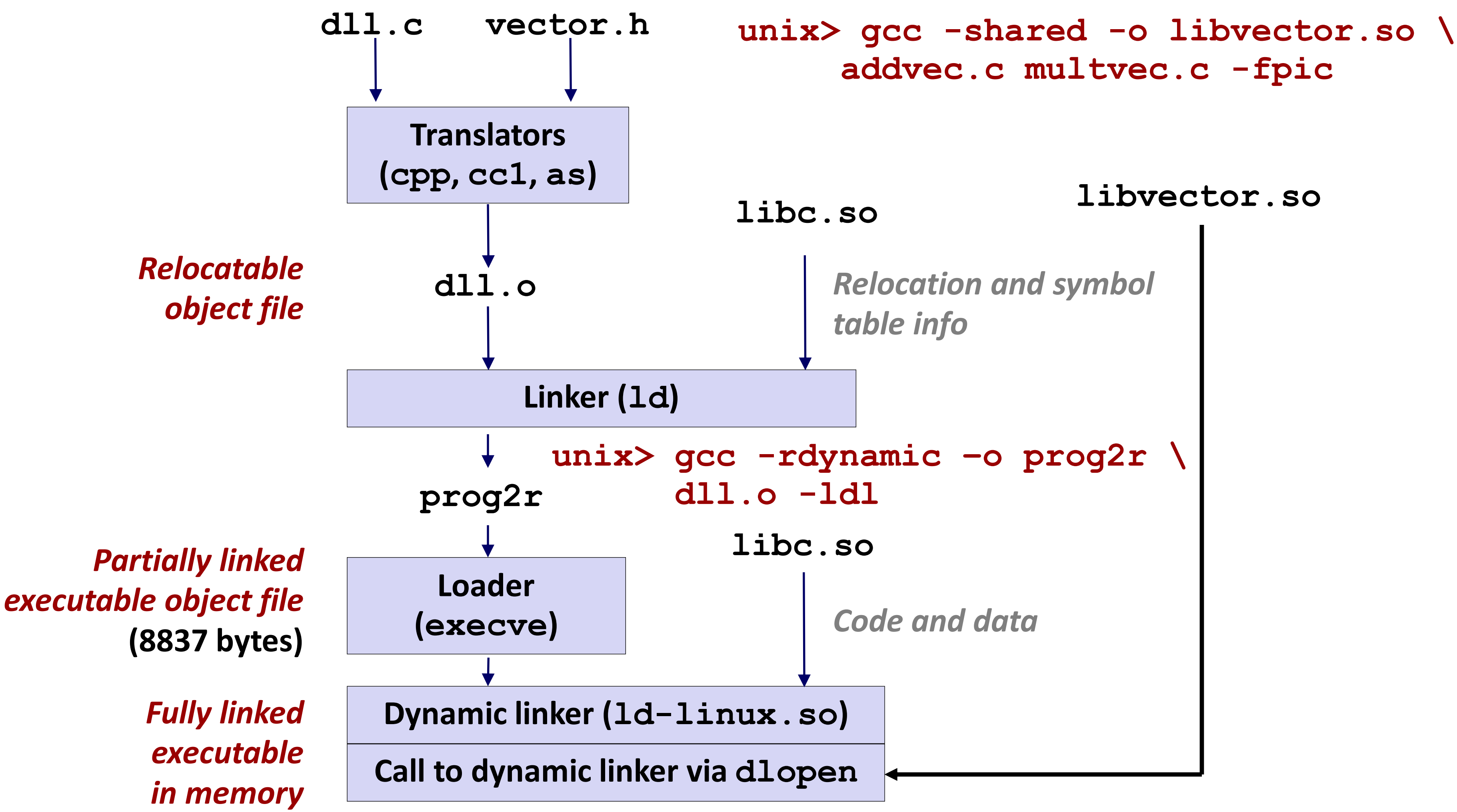
dll.c

Dynamic Linking at Run-time (cont)

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Dynamic Linking at Run-time



Linking Summary

- **Linking is a technique that allows programs to be constructed from multiple object files.**

- **Linking can happen at different times in a program's lifetime:**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

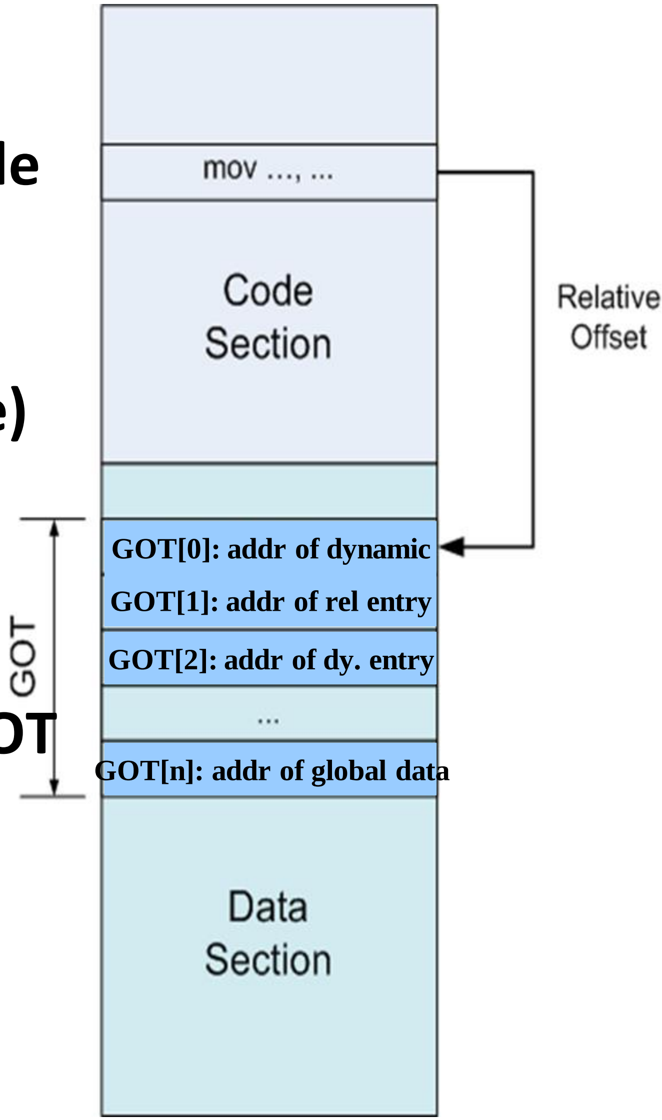
- **Understanding linking can help you avoid nasty errors and make you a better programmer.**

Today

- Linking
- **Position-Independent Code （PIC）**

PIC Data References

- A Global Offset Table(GOT) is simply a table of addresses, residing in the data section
- The GOT contains an 8-byte entry for each global object (procedure or global variable) that is referenced by the object module
- The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object.
- Each object module that references global objects has its own GOT.



Using the GOT to reference a global variable

- **The addvec routine in libvector.so references addcnt indirectly through the GOT for libvector.so**

Data segment

Global offset table (GOT)

```
GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: &addcnt
```

Fixed distance of `0x2008b9` bytes at run time between `GOT[3]` and `addl` instruction.

Code segment

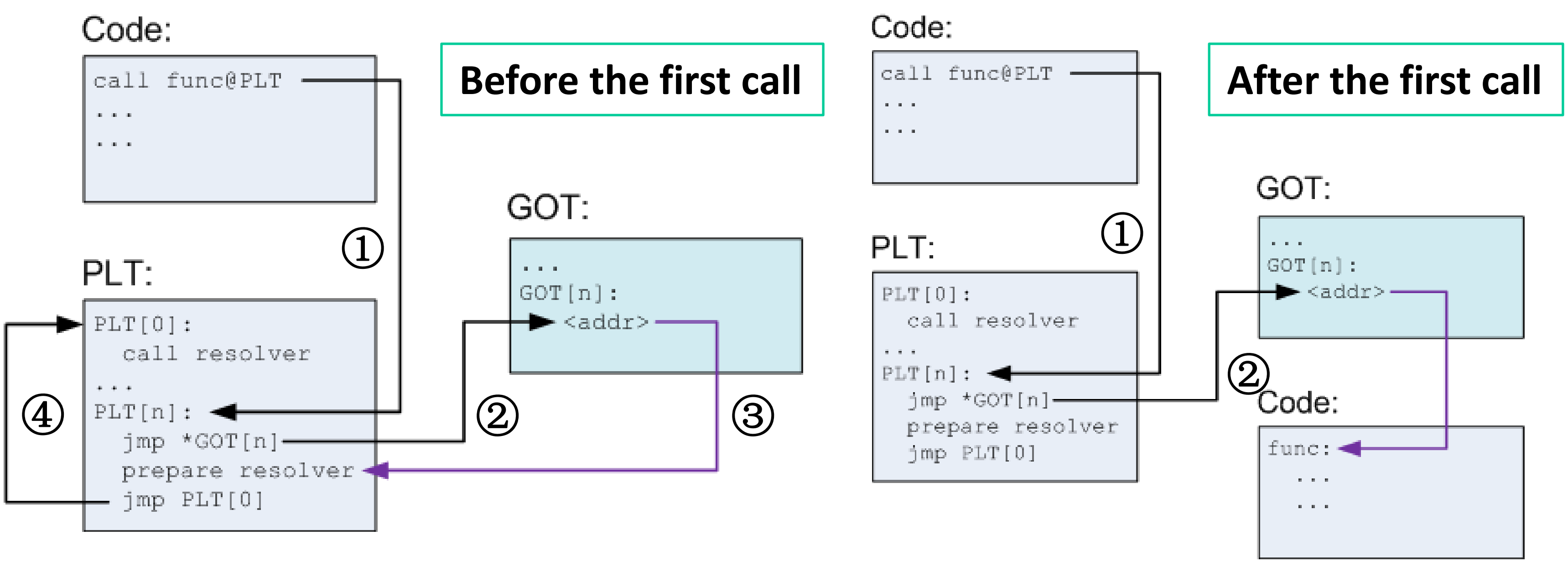
```
addvec:
  mov 0x2008b9(%rip),%rax   # %rax=*GOT[3]=&addcnt
  addl $0x1,(%rax)          # addcnt++
```

PIC Function Calls

- At load time, Dynamic linker completes the linking task by loading the shared library and relocating the references in the program.
- This task takes a nontrival time because a typical application program will call only dozens of functions exported by a shared library such as libc.so.
- Lazy binding defers the binding of each procedure address until the first time the procedure is called.
- There is a nontrivial run-time overhead the first time the function is called, but each call thereafter takes only little time
- This approch reduce the load time of a application program.
- Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the PLT.

The Procedure Linkage Table (PLT)

- The PLT is an array of 16-byte of code entries. PLT[0] is a special entry that jumps into the dynamic linker. Each shared library function called by the executables has its own PLT entry.
- Each PLT entry also has a corresponding entry in the GOT which contains the actual offset to the function, but only when the dynamic linker resolves it



Using the PLT and GOT to call external functions

- First invocation of addvec
- The dynamic linker uses the two stack entries to determine the run-time location of addvec,overwrites GOT[4] with this address, and passes control to addvec.

Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6   # sys startup
GOT[4]: 0x4005c6   # addvec()
GOT[5]: 0x4005d6   # printf()
```

Code segment

```
callq 0x4005c0   # call addvec()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

Using the PLT and GOT to call external functions

- Subsequent invocation of addvec
- Control passes to PLT[2] as before
- The indirect jump through GOT[4] transfers control directly to addvec

Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6   # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6   # printf()
```

Code segment

```
callq 0x4005c0    # call addvec()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

Example program

```
/* fopen.c
   Open a file, write "Hello World!" to it */

#include <stdio.h>
int main() {
    FILE *out;
    char buf[16] = "Hello World!\n";

    out = fopen("hello.txt", "w+");
    fprintf(out, "%s", buf);
    fclose(out);
    return 0;
}
```

PLT

```
/* section .plt */
# PLT[0] <fclose@plt-0x10>: call dynamic linker
  4004c0:  pushq  0x200b42(%rip)        # GOT[1]
  4004c6:  jmpq   *0x200b44(%rip)       # GOT[2]
  4004cc:  nopl   0x0(%rax)
# PLT[1] <fclose@plt>:
  4004d0:  jmpq   *0x200b42(%rip)       # GOT[3]
  4004d6:  pushq  $0x0
  4004db:  jmpq   4004c0 <_init+0x28>
# PLT[2] <__stack_chk_fail@plt>:
  4004e0:  jmpq   *0x200b3a(%rip)       # GOT[4]
  4004e6:  pushq  $0x1
  4004eb:  jmpq   4004c0 <_init+0x10>
# PLT[3] <fputs@plt>:
  4004f0:  jmpq   *0x200b32(%rip)       # GOT[5]
  4004f6:  pushq  $0x2
  4004fb:  jmpq   4004c0 <_init+0x10>
# PLT[4] <__libc_start_main@plt>:
  400500:  jmpq   *0x200b2a(%rip)       # GOT[6]
  400506:  pushq  $0x3
  40050b:  jmpq   4004c0 <_init+0x10>
```

GOT

Before the first call

```
/* (gdb) x /8xg 0x601000
   <_GLOBAL_OFFSET_TABLE_> */
0x601000 0x0000000000600e28  # GOT[0] addr of .dynamic
0x601008 0x00007ffff7ffe168  # GOT[1] addr of reloc entries
0x601010 0x00007ffff7deee10  # GOT[2] addr of dynamic linker
0x601018 0x00000000004004d6  # GOT[3] fclose()
0x601020 0x00000000004004e6  # GOT[4] stack_check_fail
0x601028 0x00000000004004f6  # GOT[5] fputs()
0x601030 0x00007ffff7a2d740  # GOT[6] sys startup
0x601038 0x0000000000400516  # GOT[7] fopen()
```

After the first call

```
/* (gdb) x /8xg 0x601000
   <_GLOBAL_OFFSET_TABLE_> */
0x601960 0x0000000000600e28  # GOT[0] addr of .dynamic
0x601968 0x00007ffff7ffe168  # GOT[1] addr of reloc entries
0x601970 0x00007ffff7deee10  # GOT[2] addr of dynamic linker
0x601978 0x00007ffff7a7a260  # GOT[3] fclose()
0x601980 0x00000000004004e6  # GOT[4] stack_check_fail
0x601988 0x00007ffff7a7b030  # GOT[5] fputs()
0x601990 0x00007ffff7a2d740  # GOT[6] sys startup
0x601998 0x00007ffff7a7ad70  # GOT[7] fopen()
```

教材阅读

- 第7章 **7.1-7.5、7.6.1-7.6.2、7.7、7.8、7.9、7.10、7.11、7.12**