链接与加载选讲

王慧妍
why@nju.edu.cn

南京大学 计算机科学与技术系 计算机软件研究所

提醒

PA2 Deadline 即将截止：

2023.11.19 23:59:59

推迟一周：2023.11.26 23:59:59

Lab2 Deadline 即将截止：

2023.11.19 23:59:59

推迟一周：2023.11.26 23:59:59

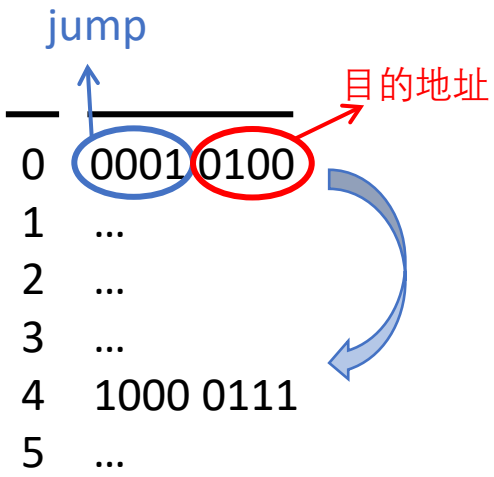
本讲概述

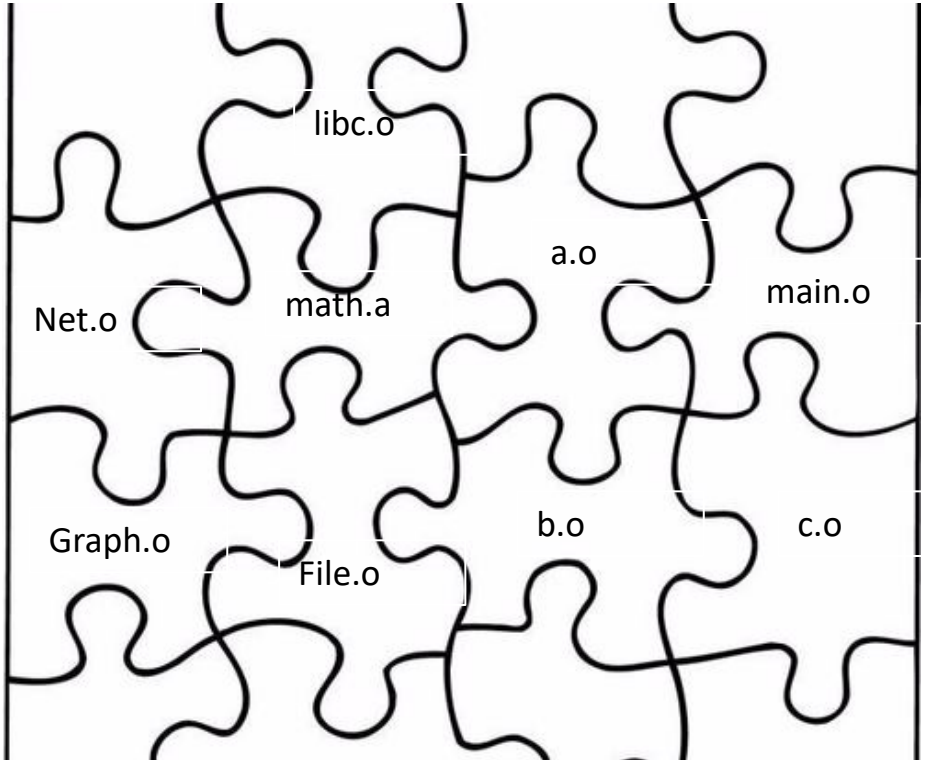
动态链接，大家明白了吗？
(根据我们的经验，大家上课没听懂)

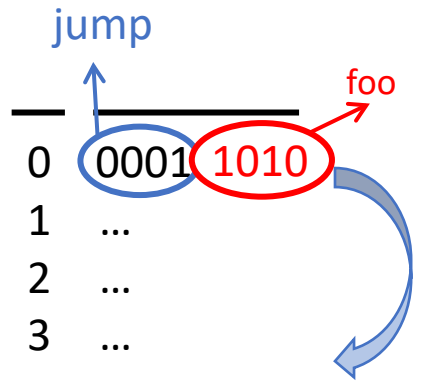
- 本讲内容
  - 静态链接与加载
    - Hello 程序的链接与加载
  - 动态链接与加载
    - 自己动手实现动态加载

静态链接与加载

什么是链接?







foo

10	1000 0111
11	...

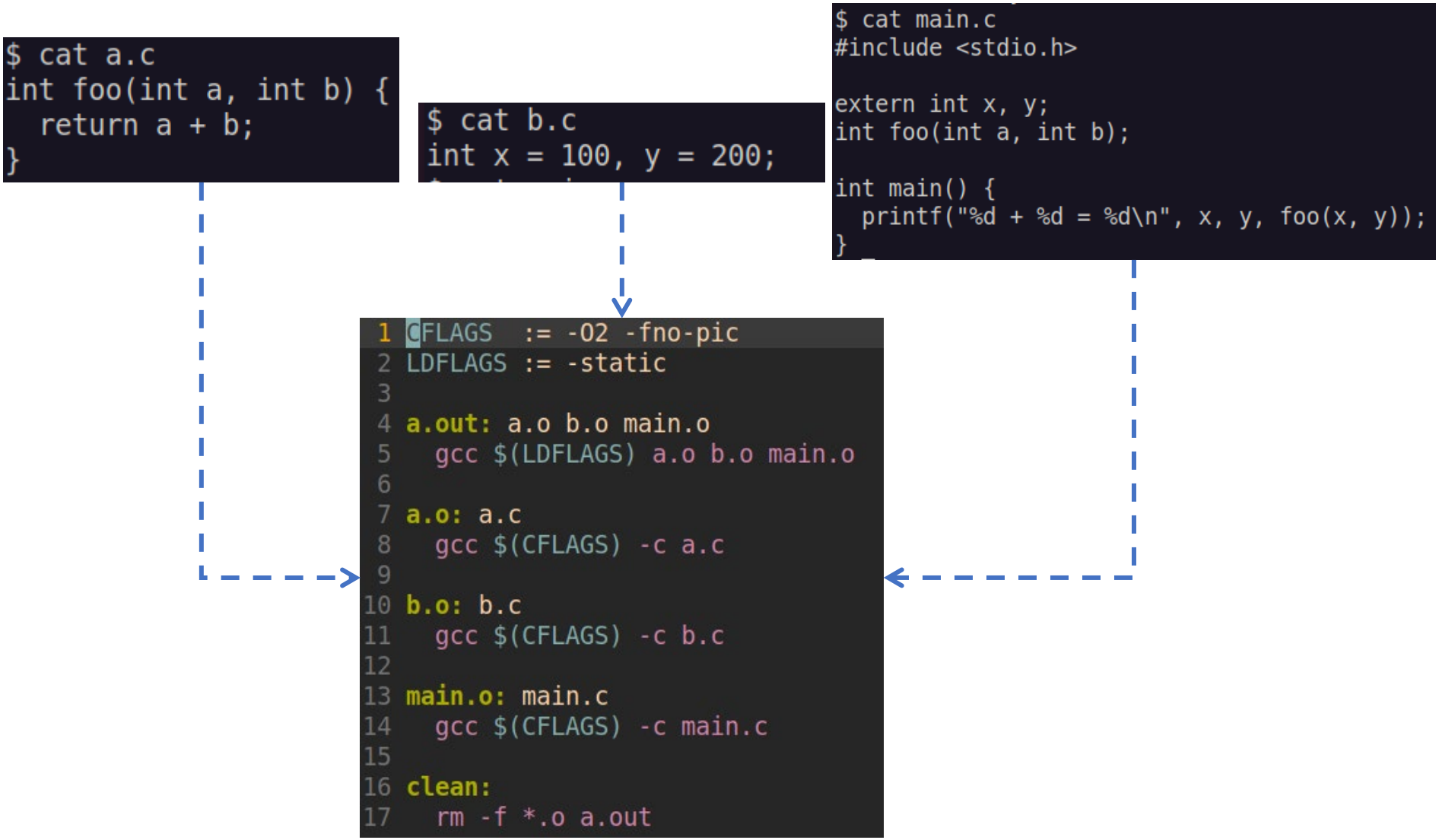
一个实验：-fno-pic编译；-static链接

- 代码

```c
// a.c
int foo(int a, int b) {
    return a + b;
}
```

```c
// b.c
int x = 100, y = 200;
```

```c
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```


$ cat a.c
int foo(int a, int b) {
return a + b;
}
$ cat b.c
int x = 100, y = 200;
$ cat main.c
#include <stdio.h>
extern int x, y;
int foo(int a, int b);
int main() {
printf("%d + %d = %d\n", x, y, foo(x, y));
}
1 CFLAGS := -O2 -fno-pic
2 LDFLAGS := -static
3
4 a.out: a.o b.o main.o
5 gcc $(LDFLAGS) a.o b.o main.o
6
7 a.o: a.c
8 gcc $(CFLAGS) -c a.c
9
10 b.o: b.c
11 gcc $(CFLAGS) -c b.c
12
13 main.o: main.c
14 gcc $(CFLAGS) -c main.c
15
16 clean:
17 rm -f *.o a.out

链接的a.o, b.o, main.o是什么?

- ELF relocatable

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

ELF文件

ELF文件类型	说明	实例
可重定位文件 (Relocatable File)	这类文件包括代码和数据，可以被用来链接成可执行文件或共享目标文件，静态链接库也可归为这一类	Linux的.o Windows的.obj
可执行文件 (Executable File)	这类文件包含了可以直接执行的程序，它的代表就是ELF可执行文件	/bin/bash，a.out Windows的.exe
共享目标文件 (Shared Object File)	这种文件包含了代码和数据，可以在下面两种情况下使用： 1. 链接器使用此文件和其他可重定位文件和共享目标文件链接，产生新的目标文件； 2. 动态链接器将几种共享目标文件与可执行文件结合，作为进程映像的一部分运行	Linux的.so（e.g.，/lib/glibc-2.5.so） Windows的DLL
核心转储文件 (Core Dump File)	当进程意外终止时，系统可以讲该进程地址空间的内容及终止时的一些信息转储到此	Linux下的core dump

可执行文件格式之EL

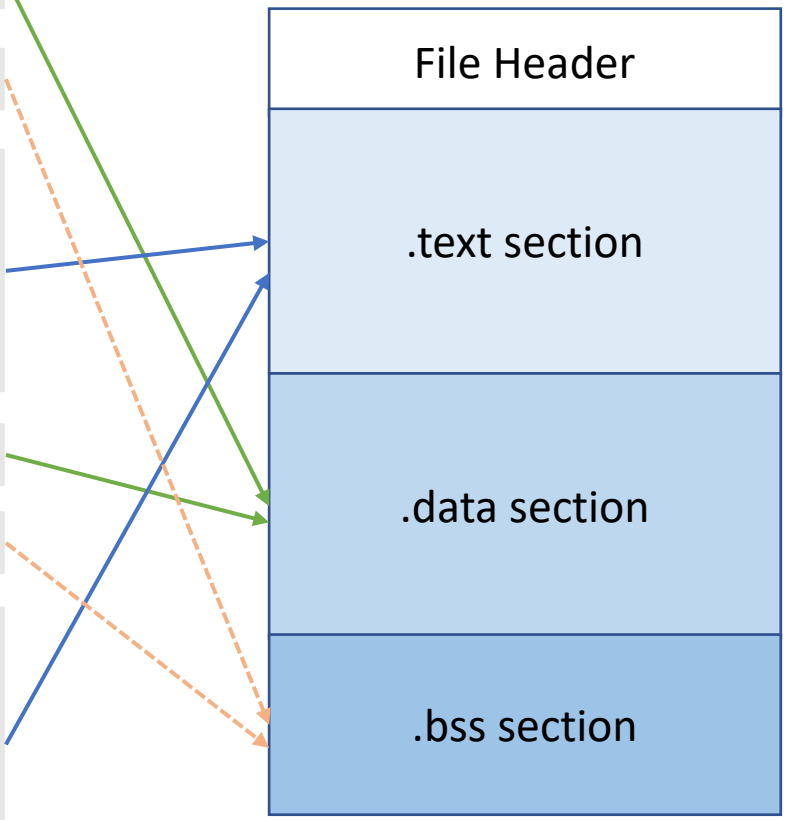
`-fno-zero-initialized-in-bss`

If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section—e.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

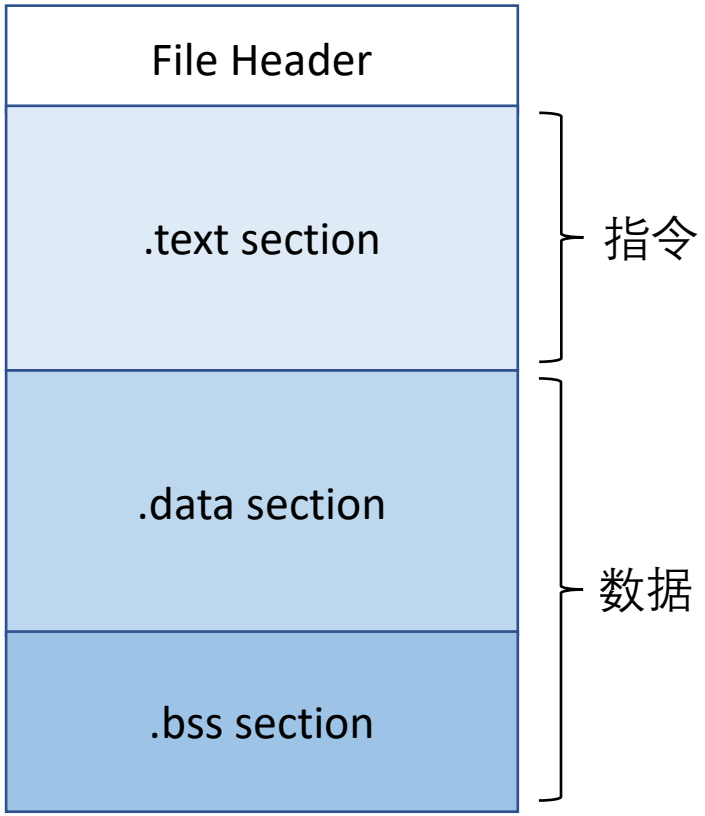
The default is '`-fzero-initialized-in-bss`'.

```
int global_int_var = 84;

int global_int_var2;

void func1(int i){
     printf("%d\n", i);
}
int main(void){

     static int static_var = 85;

     static int static_var2;

     int a = 1;
     int b;
     func1( static_var + static_var2
     + a + b);
     return 0;
}
```



小问题?

- 为什么要把程序指令和数据分开放？而不放在一个段中?
  - 权限隔离
  - 缓存命中
  - 共享内存



GNU Binutils

- Binary utilities: 分析二进制文件的工具
  - RTFM: 原来有那么多工具!
    - 有 addr2line，自己也可以实现类 gdb 调试器了

- 使用binutils
  - objdump查看.data节的x，y（-D）
  - objdump查看main对应的汇编代码
  - readelf查看relocation信息
    - 思考：为什么要-4?

GNU Binutils

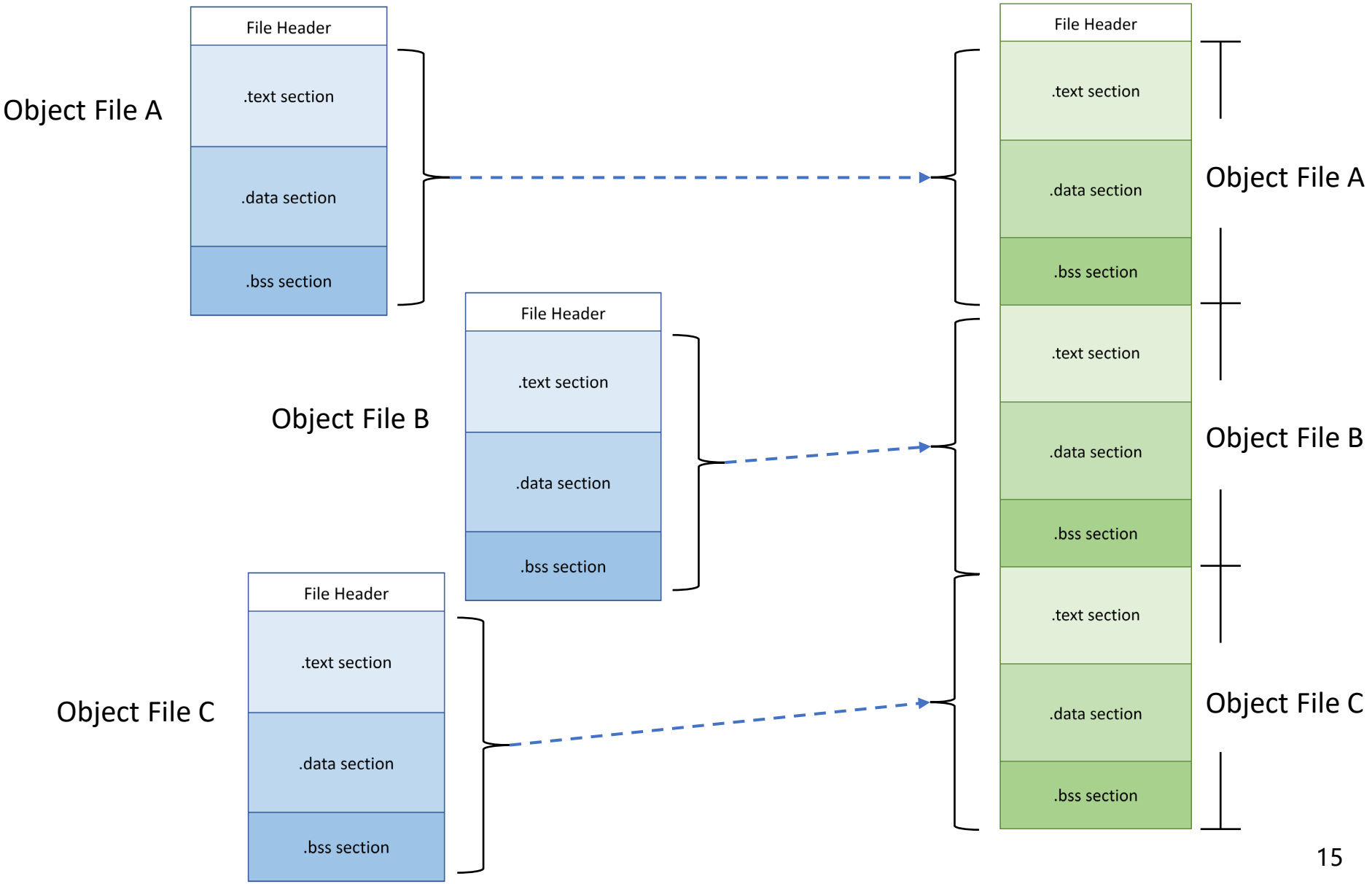
The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

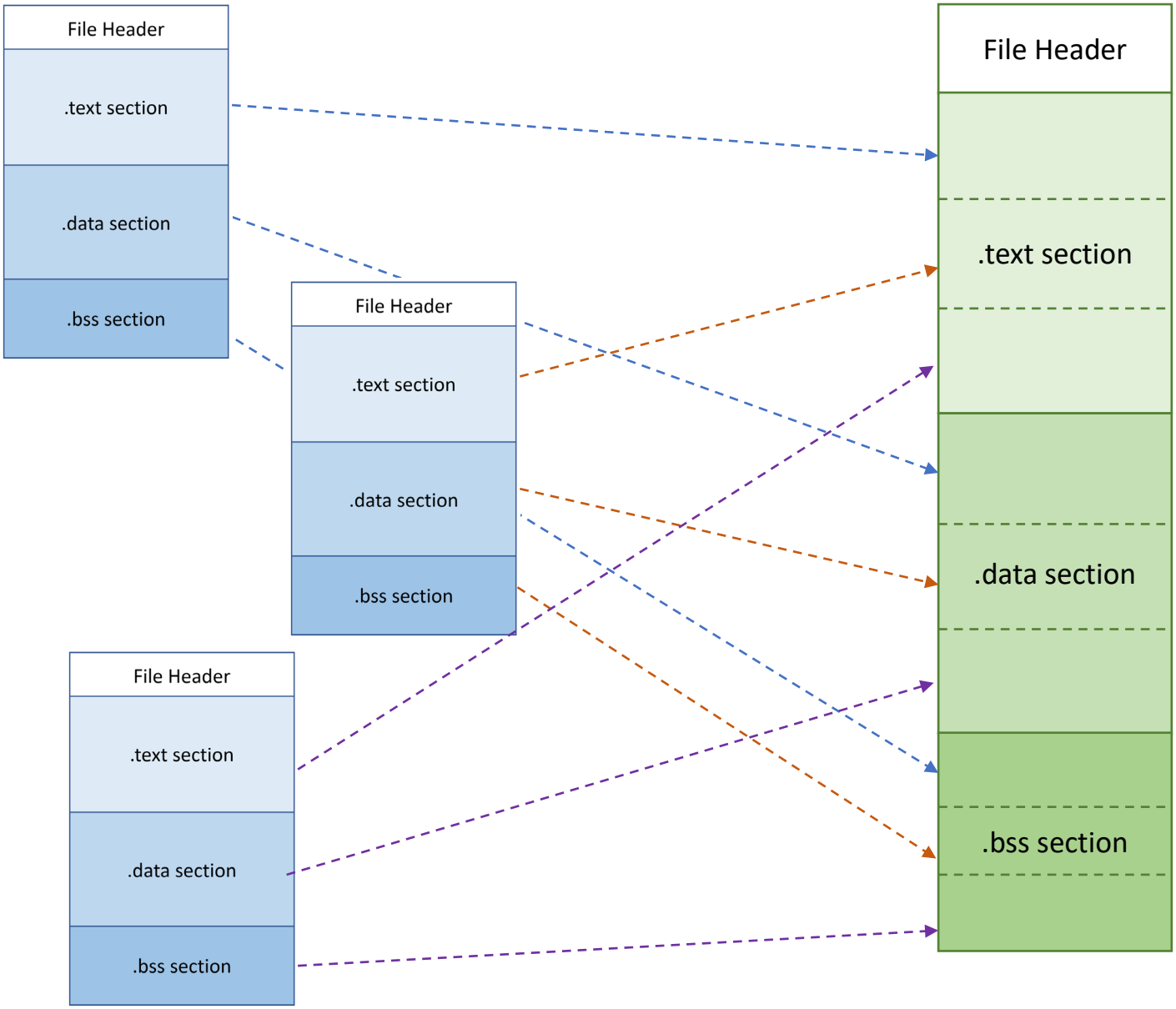
But they also include:

- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.
- **windmc** - A Windows compatible message compiler.
- **windres** - A compiler for Windows resource files.

链接多个.o


Object File A
File Header
.text section
.data section
.bss section
Object File B
File Header
.text section
.data section
.bss section
Object File C
File Header
.text section
.data section
.bss section
File Header
.text section
.data section
.bss section
Object File A
.text section
.data section
.bss section
Object File B
.text section
.data section
.bss section
Object File C

链接多个.o


File Header
.text section
.data section
.bss section
File Header
.text section
.data section
.bss section
File Header
.text section
.data section
.bss section
File Header
.text section
.data section
.bss section

一个实验：-c编译；ld链接

- 简化一点（不要printf）

```c
// a.c
int foo(int a, int b) {
   return a + b;
}
```

```c
// b.c
int x = 100, y = 200;
```

```c
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
   foo(x,y);
   //printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

Trv一下

管理 控制 视图 热键 设备 帮助

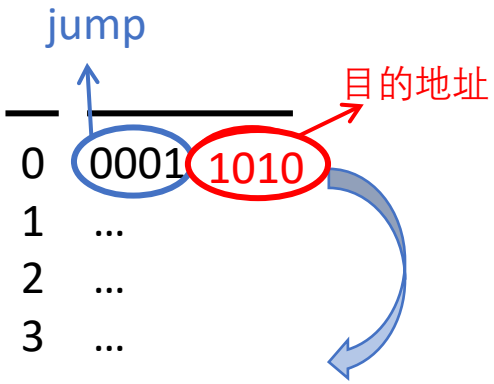
$

Right Shift + Right Alt

链接 Two-pass linking

- 空间和地址的分配
  - 重新建立符号表，合并段，并计算段长度建立映射关系

- 符号解析和重定位
  - 如何填空?



10	1000 0111
11	…

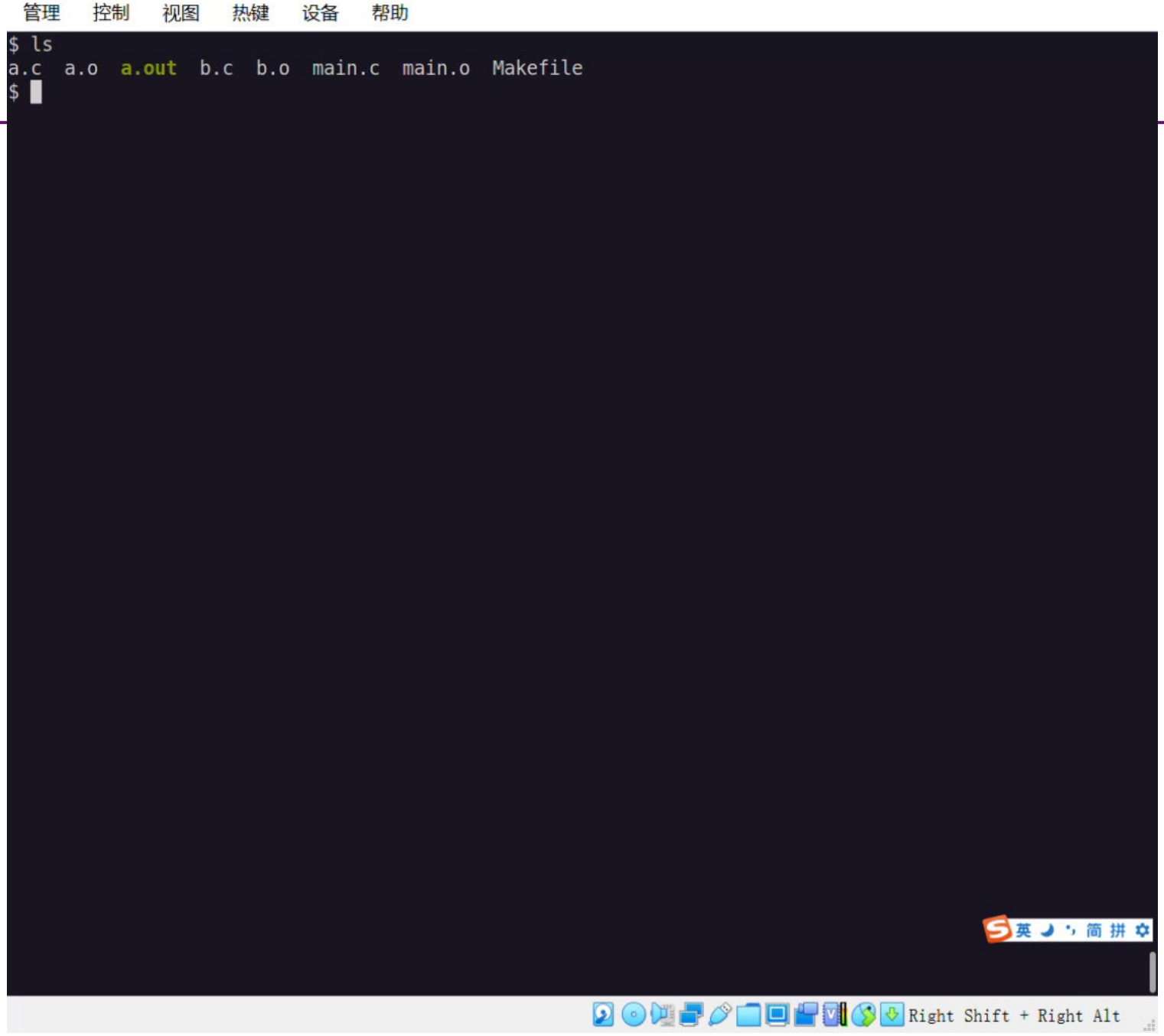
一个实验：-fno-pic编译；-static链接

- 代码

```c
// a.c
int foo(int a, int b) {
   return a + b;
}
```

```c
// b.c
int x = 100, y = 200;
```

```c
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
   printf("%d + %d = %d\n", x, y, foo(x, y));
}
```


管理 控制 视图 热键 设备 帮助
$ ls
a.c a.o a.out b.c b.o main.c main.o Makefile
$
英 简 拼
Right Shift + Right Alt

a.c / a.o

```c
// a.c
int foo(int a, int b) {
   return a + b;
}
```

```
Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   8d 04 37                lea    (%rdi,%rsi,1),%eax
   7:   c3                      ret
```

b.c / b.o

```c
// b.c
int x = 100, y = 200;
```

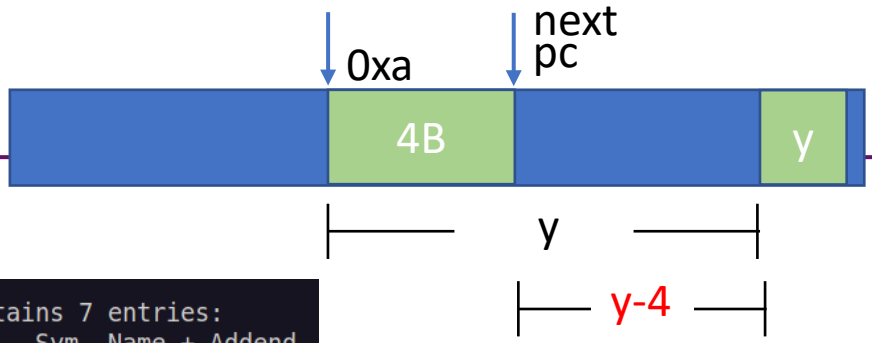
```
Disassembly of section .data:

0000000000000000 <y>:
   0:   c8 00 00 00             enter  $0x0,$0x0

0000000000000004 <x>:
   4:   64 00 00                add    %al,%fs:(%rax)
        ...
```

main.c / main.o

```c
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

```
Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   48 83 ec 08             sub    $0x8,%rsp
   8:   8b 35 00 00 00 00       mov    0x0(%rip),%esi        # e <main+0xe>
   e:   8b 3d 00 00 00 00       mov    0x0(%rip),%edi        # 14 <main+0x14>
  14:   e8 00 00 00 00          call   19 <main+0x19>
  19:   8b 0d 00 00 00 00       mov    0x0(%rip),%ecx        # 1f <main+0x1f>
  1f:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 25 <main+0x25>
  25:   be 00 00 00 00          mov    $0x0,%esi
  2a:   41 89 c0                mov    %eax,%r8d
  2d:   bf 01 00 00 00          mov    $0x1,%edi
  32:   31 c0                   xor    %eax,%eax
  34:   e8 00 00 00 00          call   39 <main+0x39>
  39:   31 c0                   xor    %eax,%eax
  3b:   48 83 c4 08             add    $0x8,%rsp
  3f:   c3                      ret
```

填什么？为什么y - 4？

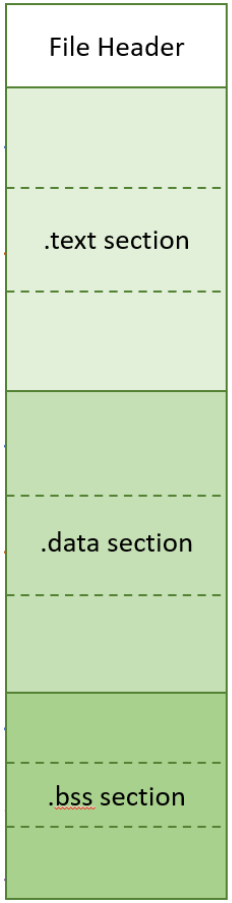


- readelf –a main.o

```
Relocation section '.rela.text.startup' at offset 0x208 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  000500000002 R_X86_64_PC32     0000000000000000 y - 4
000000000010  000600000002 R_X86_64_PC32     0000000000000000 x - 4
000000000015  000700000004 R_X86_64_PLT32    0000000000000000 foo - 4
00000000001b  000500000002 R_X86_64_PC32     0000000000000000 y - 4
000000000021  000600000002 R_X86_64_PC32     0000000000000000 x - 4
000000000026  00020000000a R_X86_64_32       0000000000000000 .rodata.str1.1 + 0
000000000035  000800000004 R_X86_64_PLT32    0000000000000000 __printf_chk - 4
```

- objdump –d main.o

```
Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   48 83 ec 08             sub    $0x8,%rsp
   8:   8b 35 00 00 00 00       mov    0x0(%rip),%esi        # e <main+0xe>
   e:   8b 3d 00 00 00 00       mov    0x0(%rip),%edi        # 14 <main+0x14>
  14:   e8 00 00 00 00          call   19 <main+0x19>
  19:   8b 0d 00 00 00 00       mov    0x0(%rip),%ecx        # 1f <main+0x1f>
  1f:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 25 <main+0x25>
  25:   be 00 00 00 00          mov    $0x0,%esi
  2a:   41 89 c0                mov    %eax,%r8d
  2d:   bf 01 00 00 00          mov    $0x1,%edi
  32:   31 c0                   xor    %eax,%eax
  34:   e8 00 00 00 00          call   39 <main+0x39>
  39:   31 c0                   xor    %eax,%eax
  3b:   48 83 c4 08             add    $0x8,%rsp
  3f:   c3                      ret
```



填什么?

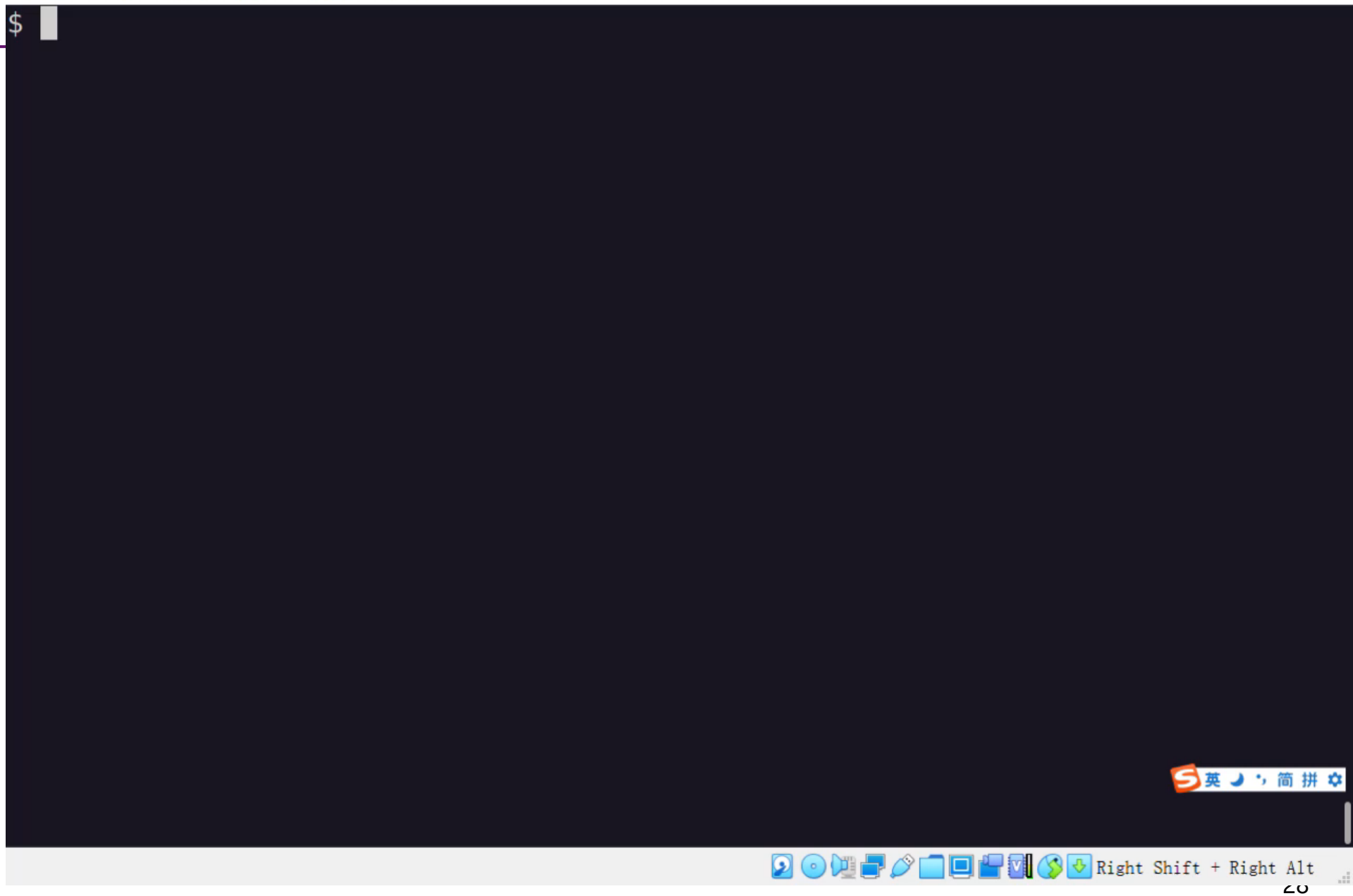
- objdump –d a.out

```
0000000000401790 <main>:
  401790:       f3 0f 1e fa             endbr64
  401794:       48 83 ec 08             sub    $0x8,%rsp
  401798:       8b 35 52 89 0b 00       mov    0xb8952(%rip),%esi        # 4ba0f0 <y>
  40179e:       8b 3d 50 89 0b 00       mov    0xb8950(%rip),%edi        # 4ba0f4 <x>
  4017a4:       e8 57 01 00 00          call   401900 <foo>
  4017a9:       8b 0d 41 89 0b 00       mov    0xb8941(%rip),%ecx        # 4ba0f0 <y>
  4017af:       8b 15 3f 89 0b 00       mov    0xb893f(%rip),%edx        # 4ba0f4 <x>
  4017b5:       be 04 e0 48 00          mov    $0x48e004,%esi
  4017ba:       41 89 c0                mov    %eax,%r8d
  4017bd:       bf 01 00 00 00          mov    $0x1,%edi
  4017c2:       31 c0                   xor    %eax,%eax
  4017c4:       e8 47 3e 04 00          call   445610 <__printf_chk>
  4017c9:       31 c0                   xor    %eax,%eax
  4017cb:       48 83 c4 08             add    $0x8,%rsp
  4017cf:       c3                      ret
```

填什么?

```
0000000000401790 <main>:
  401790:       f3 0f 1e fa             endbr64
  401794:       48 83 ec 08             sub    $0x8,%rsp
  401798:       8b 35 52 89 0b 00       mov    0xb8952(%rip),%esi        # 4ba0f0 <y>
  40179e:       8b 3d 50 89 0b 00       mov    0xb8950(%rip),%edi        # 4ba0f4 <x>
  4017a4:       e8 57 01 00 00          call   401900 <foo>
  4017a9:       8b 0d 41 89 0b 00       mov    0xb8941(%rip),%ecx        # 4ba0f0 <y>
  4017af:       8b 15 3f 89 0b 00       mov    0xb893f(%rip),%edx        # 4ba0f4 <x>
  4017b5:       be 04 e0 48 00          mov    $0x48e004,%esi
  4017ba:       41 89 c0                mov    %eax,%r8d
  4017bd:       bf 01 00 00 00          mov    $0x1,%edi
  4017c2:       31 c0                   xor    %eax,%eax
  4017c4:       e8 47 3e 04 00          call   445610 <__printf_chk>
  4017c9:       31 c0                   xor    %eax,%eax
  4017cb:       48 83 c4 08             add    $0x8,%rsp
  4017cf:       c3                      ret
```

```
0000000000401900 <foo>:
  401900:       f3 0f 1e fa             endbr64
  401904:       8d 04 37                lea    (%rdi,%rsi,1),%eax
  401907:       c3                      ret
  401908:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40190f:       00
```


管理 控制 视图 热键 设备 帮助
$
英 简 拼
Right Shift + Right Alt

谁来加载a.out?

管理 控制 视图 热键 设备 帮助

```
0000000000401810 <main>:
  401810:       f3 0f 1e fa             endbr64
  401814:       48 83 ec 08             sub    $0x8,%rsp
  401818:       bf a0 17 40 00          mov    $0x4017a0,%edi
  40181d:       e8 de 96 00 00          call   40af00 <atexit>
  401822:       8b 35 c8 88 0b 00       mov    0xb88c8(%rip),%esi        # 4ba0
f0 <y>
  401828:       8b 3d c6 88 0b 00       mov    0xb88c6(%rip),%edi        # 4ba0
f4 <x>
  40182e:       e8 5d 01 00 00          call   401990 <foo>
  401833:       8b 0d b7 88 0b 00       mov    0xb88b7(%rip),%ecx        # 4ba0
f0 <y>
  401839:       8b 15 b5 88 0b 00       mov    0xb88b5(%rip),%edx        # 4ba0
f4 <x>
  40183f:       be 10 e0 48 00          mov    $0x48e010,%esi
  401844:       41 89 c0                mov    %eax,%r8d
  401847:       bf 01 00 00 00          mov    $0x1,%edi
  40184c:       31 c0                   xor    %eax,%eax
  40184e:       e8 4d 40 04 00          call   4458a0 <___printf_chk>
  401853:       31 c0                   xor    %eax,%eax
  401855:       48 83 c4 08             add    $0x8,%rsp
  401859:       c3                      ret
  40185a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
:
```

Right Shift + Right Alt

静态程序的加载

由操作系统加载（下学期内容）

- 你可以认为是“一步到位”的
  - 使用gdb (starti)调试
  - 使用strace查看系统调用序列

```
$ objdump -d a.out | less
$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401860
  Start of program headers:          64 (bytes into file)
  Start of section headers:          844280 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         32
```

管理 控制 视图 热键 设备 帮助

```
World
$ objdump -d a.out | less
$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401860
  Start of program headers:          64 (bytes into file)
  Start of section headers:          844280 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         32
  Section header string table index: 31
$
```

Right Shift + Right Alt

strace

```
$ make clean
rm -f *.o a.out
$ make
gcc -O2 -fno-pic -c a.c
gcc -O2 -fno-pic -c b.c
gcc -O2 -fno-pic -c main.c
gcc -static a.o b.o main.o
$ ./a.out
100 + 200 = 300
$ strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffcc0649630 /* 60 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe0d8c6370) = -1 EINVAL (Invalid argument)
brk(NULL)                               = 0x1dad000
brk(0x1dadd80)                          = 0x1dadd80
arch_prctl(ARCH_SET_FS, 0x1dad380)      = 0
uname({sysname="Linux", nodename="why-VirtualBox", ...}) = 0
readlink("/proc/self/exe", "/home/why/Documents/ICS2021/teac"..., 4096) = 49
brk(0x1dced80)                          = 0x1dced80
brk(0x1dcf000)                          = 0x1dcf000
mprotect(0x4b6000, 16384, PROT_READ)    = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "100 + 200 = 300\n", 16100 + 200 = 300
)       = 16
exit_group(0)                           = ?
+++ exited with 0 +++
```

管理 控制 视图 热键 设备 帮助

$

Right Shift + Right Alt

一个有趣的问题：“最小”的可执行代码

不能用ld链接么？

- （试一试）
  - 我们能否写一个最小的汇编代码，能正确返回？
  - 仅执行一个操作系统调用
    - `rax = 231; rdi = 返回值`
    - `syscall`指令执行系统调用

管理 控制 视图 热键 设备 帮助

```
$ vim as.S
```


Right Shift + Right Alt

gcc和ld链接

不能用ld链接么？

- man gcc

```
       to write -Xlinker "-assert definitions", because this passes the entire string as a
       single argument, which is not what the linker expects.

       When using the GNU linker, it is usually more convenient to pass arguments to linker
       options using the option=value syntax than as separate arguments.  For example, you
       can specify -Xlinker -Map=output.map rather than -Xlinker -Map -Xlinker output.map.
       Other linkers may not support this syntax for command-line options.

   -Wl,option
       Pass option as an option to the linker.  If option contains commas, it is split into
       multiple options at the commas.  You can use this syntax to pass an argument to the
       option.  For example, -Wl,-Map,output.map passes -Map output.map to the linker.  When
       using the GNU linker, you can also get the same effect with -Wl,-Map=output.map.

       NOTE: In Ubuntu 8.10 and later versions, for LDFLAGS, the option -Wl,-z,relro is used.
       To disable, use -Wl,-z,norelro.

   -u symbol
       Pretend the symbol symbol is undefined, to force linking of library modules to define
       it.  You can use -u multiple times with different symbols to force loading of
       additional library modules.

   -z keyword
       -z is passed directly on to the linker along with the keyword keyword. See the section
       in the documentation of your linker for permitted values and their meanings.

Options for Directory Search
   These options specify directories to search for header files, for libraries and for parts
   of the compiler:

   -I dir
   -iquote dir
Manual page gcc(1) line 11096/23056 53% (press h for help or q to quit)
```

管理 控制 视图 热键 设备 帮助

$

Right Shift + Right Alt

一些常见的链接库

```
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)unordtf2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)letf2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)sfp-exceptions.o
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-dw2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-dw2-fde-dip.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-c.o
/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)dl-iterateph
dr.o
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a
/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a
attempt to open /usr/lib/gcc/x86_64-linux-gnu/10/crtend.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/10/crtend.o
attempt to open /usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crtn
.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crtn.o
```

Linux/C世界的宝藏

- `gcc -Wl,--verbose`
- C世界里还有很多大家不知道的东西
  - 一系列crt对象
    - `__attribute__((constructor))`
    - `__attribute__((destructor))` (atexit)
- RTFM; RTFSC!

管理 控制 视图 热键 设备 帮助

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

extern int x, y;
int foo(int a, int b);

__attribute__((constructor)) void a(){
    printf("Hello\n");
}
__attribute__((destructor)) void b(){
    printf("World\n");
}

int main() {

    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

~/Documents/ICS2021/teach/link-test/main.c[+1] [c] unix utf-8 Ln 14, Col 13/17

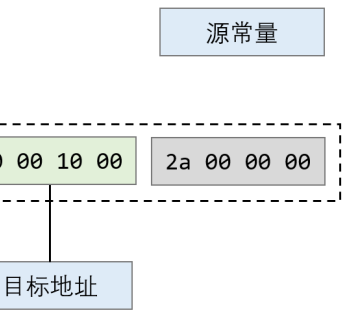
-- INSERT --

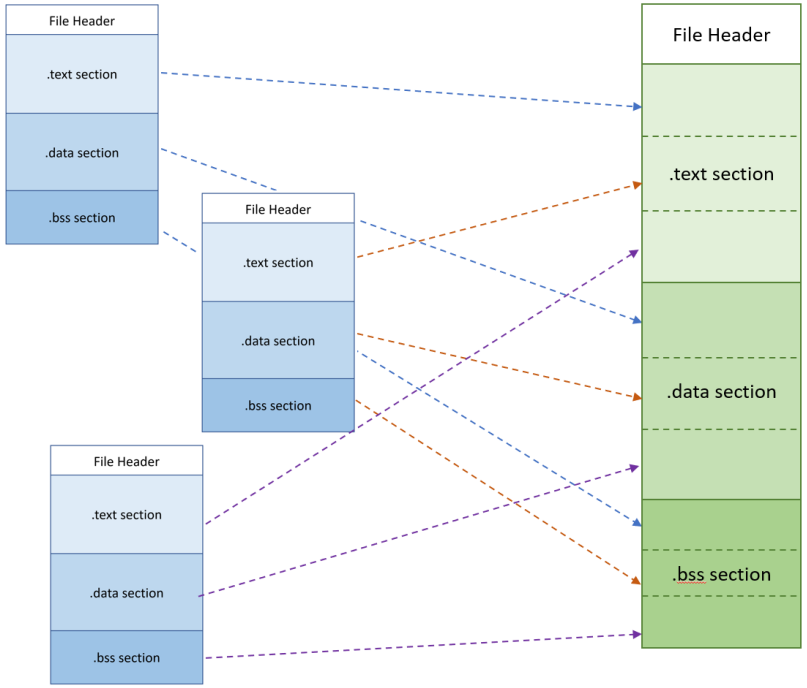
Right Shift + Right Alt

Libc.a

```
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strops.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)alloca_cutof
f.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)libc-lowleve
llock.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)malloc.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)morecore.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strchr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strcmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strcspn.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strdup.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strlen.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strncmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strstr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)memcmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)memmove.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)memset.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)mempcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)stpcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)strcasecmp_l
.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)memcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)rawmemchr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/libc.a)st
```

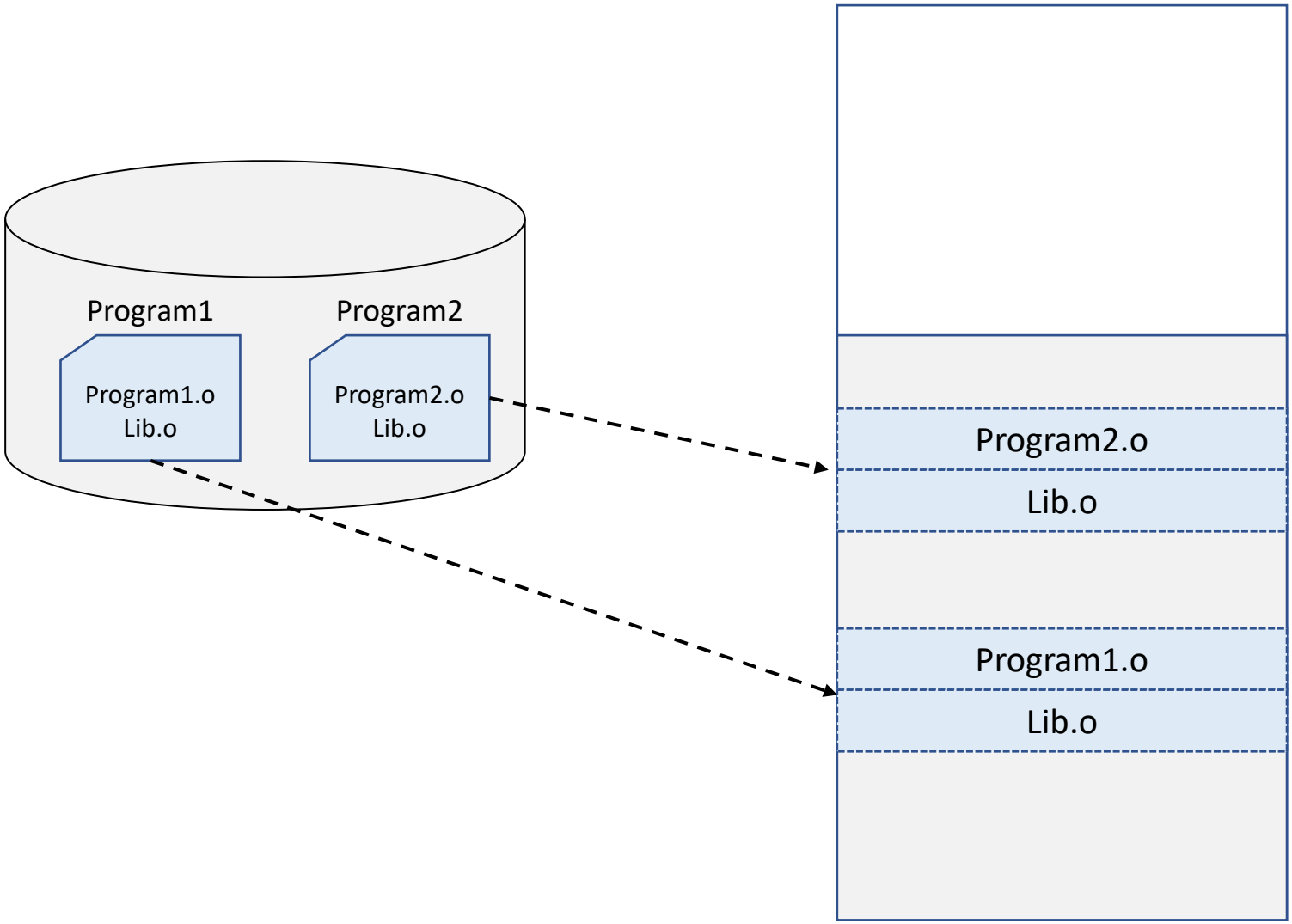
- A: 定义var (0x1000)
- B: movl $0x2a, var



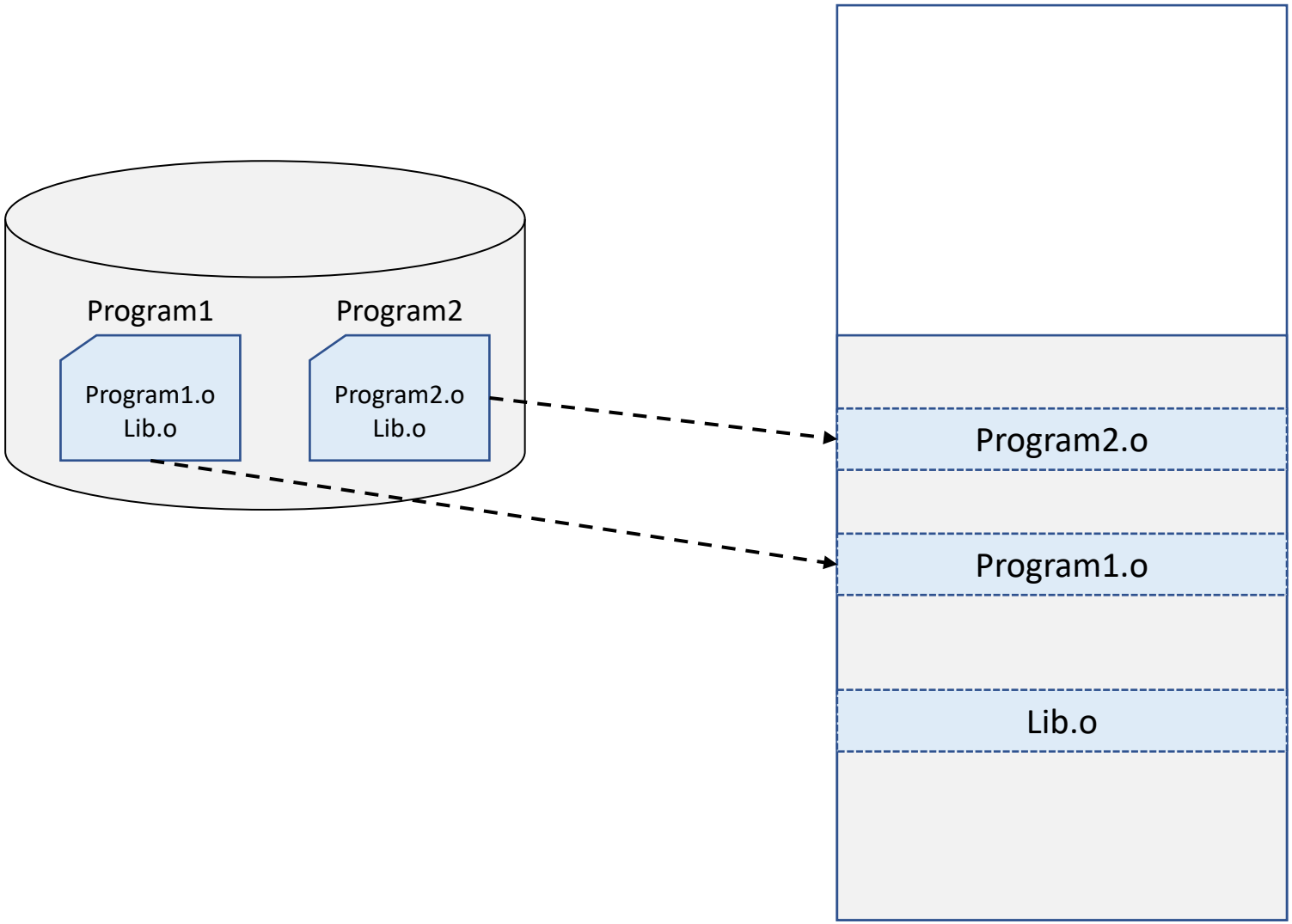


动态链接与加载

为什么要动态链接?


Program1
Program2
Program1.o
Lib.o
Program2.o
Lib.o
Program2.o
Lib.o
Program1.o
Lib.o

为什么要动态链接?


Program1
Program2
Program1.o
Lib.o
Program2.o
Lib.o
Program2.o
Program1.o
Lib.o

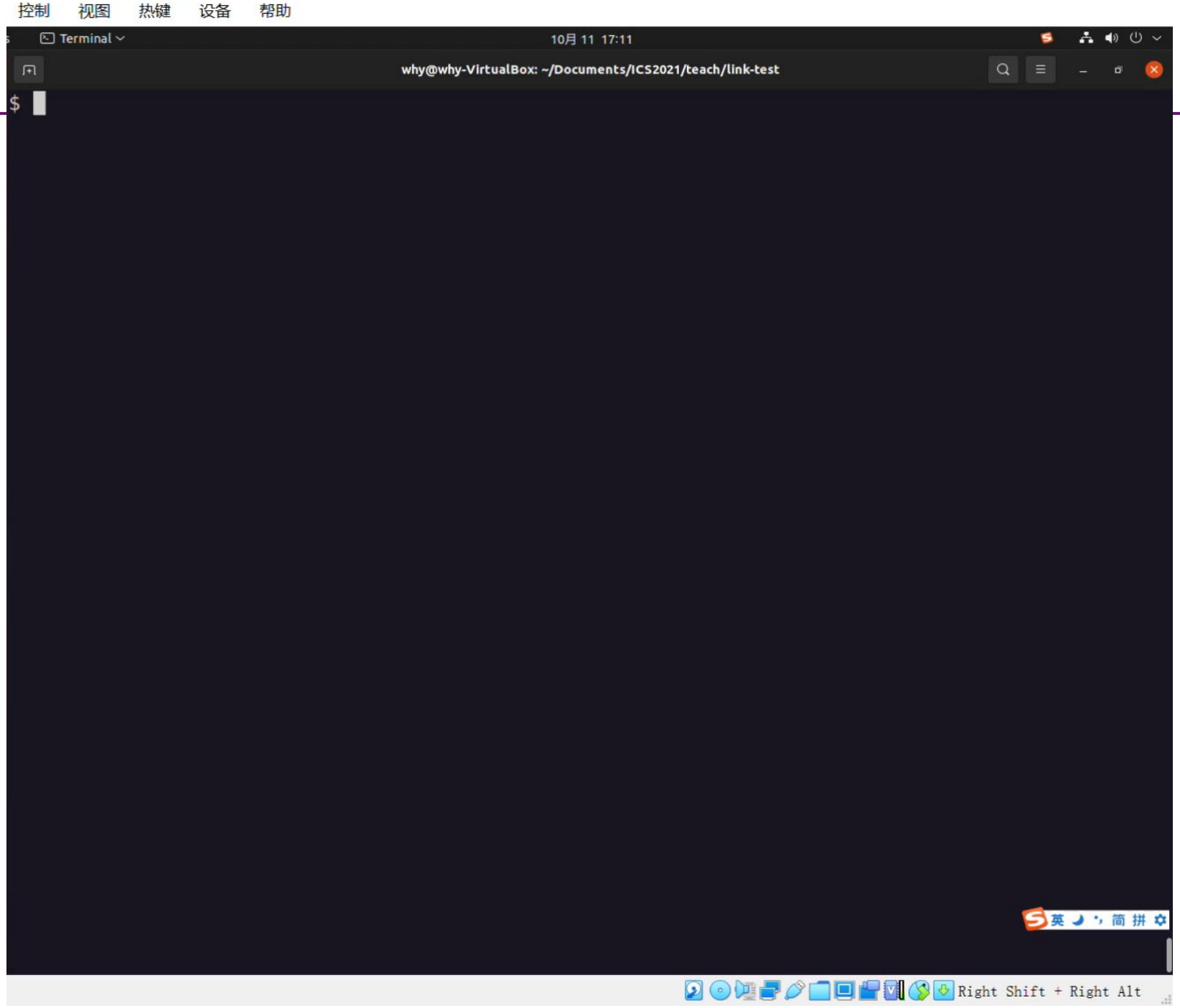
为什么要动态链接?

```
$ ls
a.c  a.o  a.out  b.c  b.o  main.c  main.o  Makefile
$ ls -l
total 856
-rw-rw-r-- 1 why why     42 11月  3  2020 a.c
-rw-rw-r-- 1 why why   1200  9月 28 10:29 a.o
-rwxrwxr-x 1 why why 846328  9月 28 10:37 a.out
-rw-rw-r-- 1 why why     22 11月  3  2020 b.c
-rw-rw-r-- 1 why why    984  9月 28 10:29 b.o
-rw-rw-r-- 1 why why    278  9月 28 10:37 main.c
-rw-rw-r-- 1 why why   2752  9月 28 10:37 main.o
-rw-rw-r-- 1 why why    241  9月 28 10:25 Makefile
```

为什么要动态链接

- 去掉-fno-pic和-static

```
$ make
gcc  -c a.c
gcc  -c b.c
gcc  -c main.c
gcc  a.o b.o main.o
$ ls -l
total 48
-rw-rw-r-- 1 why why    42 11月  3  2020 a.c
-rw-rw-r-- 1 why why  1224  9月 28 10:47 a.o
-rwxrwxr-x 1 why why 16416  9月 28 10:47 a.out
-rw-rw-r-- 1 why why    22 11月  3  2020 b.c
-rw-rw-r-- 1 why why   984  9月 28 10:47 b.o
-rw-rw-r-- 1 why why   278  9月 28 10:37 main.c
-rw-rw-r-- 1 why why  2576  9月 28 10:47 main.o
-rw-rw-r-- 1 why why   245  9月 28 10:47 Makefile
```


控制 视图 热键 设备 帮助
Terminal
10月 11 17:11
why@why-VirtualBox: ~/Documents/ICS2021/teach/link-test
$
英 简 拼
Right Shift + Right Alt

去掉-fno-pic和-static

这是默认的gcc编译/链接选项

- 文件相比静态链接大幅瘦身
  - a.o, b.o没有变化
  - main.o里面依然有00 00 00 00
    - 但是相对于rip的offset
    - relocation依然是x - 4, y - 4, foo - 4
  - a.out里面库的代码都不见了

位置无关代码

- 使用位置无关代码（PIC）的原因
  - 共享库不必事先决定加载的位置
  - 应用程序自己也是
    - 新版本gcc无论32/64 bit均默认PIC
    - 重现课本行为可以使用-no-pie选项编译

- PIC的实现
  - i386并不支持以下代码

```
movl $1, 1234(%eip)
```

  - 于是有了你们经常看到的__i686.get_pc_thunk.bx
    - “获取 next PC 的地址”(如何实现?)

```
mov (%esp), %ebx
```

动态链接

./a.out的执行

```
$ vim $(which ldd)
```

```
$ vim $(which ldd)
$ /lib64/ld-linux-x86-64.so.2 --list a.out
a.out: error while loading shared libraries: a.out: cannot open shared object file
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
        linux-vdso.so.1 (0x00007ffe23928000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9727540000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9727743000)
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
        linux-vdso.so.1 (0x00007fff7ffd6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff6f3e66000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff6f4069000)
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
        linux-vdso.so.1 (0x00007ffc715df000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2042827000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2042a2a000)
$ /lib64/ld-linux-x86-64.so.2 ./a.out
Hello
100 + 200 = 300
World
World
$
```

管理 控制 视图 热键 设备 帮助

$

共享对象（a.out）的加载

- 命令：ldd
  - “print shared object dependencies”
    - linux-vdso.so.1
      - 暂时忽略它
    - libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
    - /lib64/ld-linux-x86-64.so.2
      - 多次打印，地址会发生变化（ldd会执行加载过程）
  - Ldd竟然是一个脚本（vim $(which ldd)）
    - 挨个尝试调用若干ld-linux.so候选
      - 加上一系列环境变量
      - 我们可以用--list选项来达到类似效果

共享对象（a.out）的加载（cont'd）

- 终于揭开谜题
  - `readelf -a a.out`
    - program header中有一个INTERP
      - `/lib64/ld-linux-x86-64.so.2`
      - 这个字符串可以直接在二进制文件中看到

```
$ readelf -l a.out | grep interpreter
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

- 神奇的ld.so
  - `/lib64/ld-linux-x86-64.so.2 ./a.out`
    - 和执行`./a.out`的行为完全一致
      - （这才是`./a.out`真正在操作系统里发生的事情）
      - 与sha-bang(#!)实现机制完全相同
  - `ld.so`到底做了什么？（下学期分解）

动态链接器/lib/ld-linux.so.2

- 动态链接器本身是动态还是静态链接的，为什么?

```
$ ldd /lib64/ld-linux-x86-64.so.2
        statically linked
```

- 动态链接器是否是PIC的?

```
$ readelf -d /lib64/ld-linux-x86-64.so.2

Dynamic section at offset 0x32e70 contains 19 entries:
  Tag        Type                         Name/Value
 0x000000000000000e (SONAME)             Library soname: [ld-linux-x86-64.so.2]
 0x0000000000000004 (HASH)               0x2f0
 0x000000006ffffef5 (GNU_HASH)           0x3b8
 0x0000000000000005 (STRTAB)             0x790
 0x0000000000000006 (SYMTAB)             0x4a8
 0x000000000000000a (STRSZ)              549 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000003 (PLTGOT)             0x34000
 0x0000000000000002 (PLTRELSZ)           96 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0xb90
 0x0000000000000007 (RELA)               0xaa0
 0x0000000000000008 (RELASZ)             240 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffc (VERDEF)             0x9f8
 0x000000006ffffffd (VERDEFNUM)          5
 0x000000006ffffff0 (VERSYM)             0x9b6
 0x000000006ffffff9 (RELACOUNT)          8
 0x0000000000000000 (NULL)               0x0
```

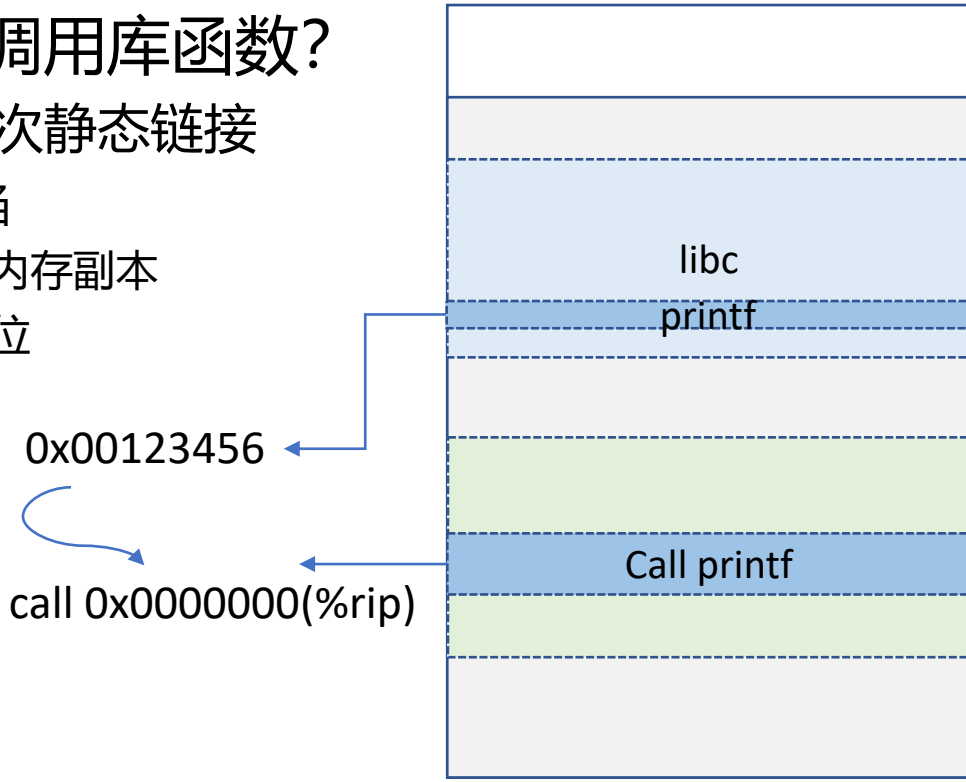
动态链接：实现

我们刚才忽略了一个巨大的问题

main.o在链接时，对printf的调用地址就确定了
但是libc每次加载的位置都不一样啊！

- 应用程序使用怎样的指令序列调用库函数?
  - 可以在库加载的时候重新进行一次静态链接
    - 但是这个方案有一些明显的缺陷
      - 各个进程的代码不能共享一个内存副本
      - 没有使用过的符号也需要重定位

```
000000000000002e <main>:
  2e:   f3 0f 1e fa             endbr64
  32:   55                      push   %rbp
  33:   48 89 e5                mov    %rsp,%rbp
  36:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 3d <main+0xf>
  3d:   e8 00 00 00 00          call   42 <main+0x14>
  42:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 48 <main+0x1a>
  48:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 4e <main+0x20>
  4e:   89 d6                   mov    %edx,%esi
  50:   89 c7                   mov    %eax,%edi
  52:   e8 00 00 00 00          call   57 <main+0x29>
  57:   89 c1                   mov    %eax,%ecx
  59:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 5f <main+0x31>
  5f:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 65 <main+0x37>
  65:   89 c6                   mov    %eax,%esi
  67:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 6e <main+0x40>
  6e:   b8 00 00 00 00          mov    $0x0,%eax
  73:   e8 00 00 00 00          call   78 <main+0x4a>
  78:   b8 00 00 00 00          mov    $0x0,%eax
  7d:   5d                      pop    %rbp
  7e:   c3                      ret
```

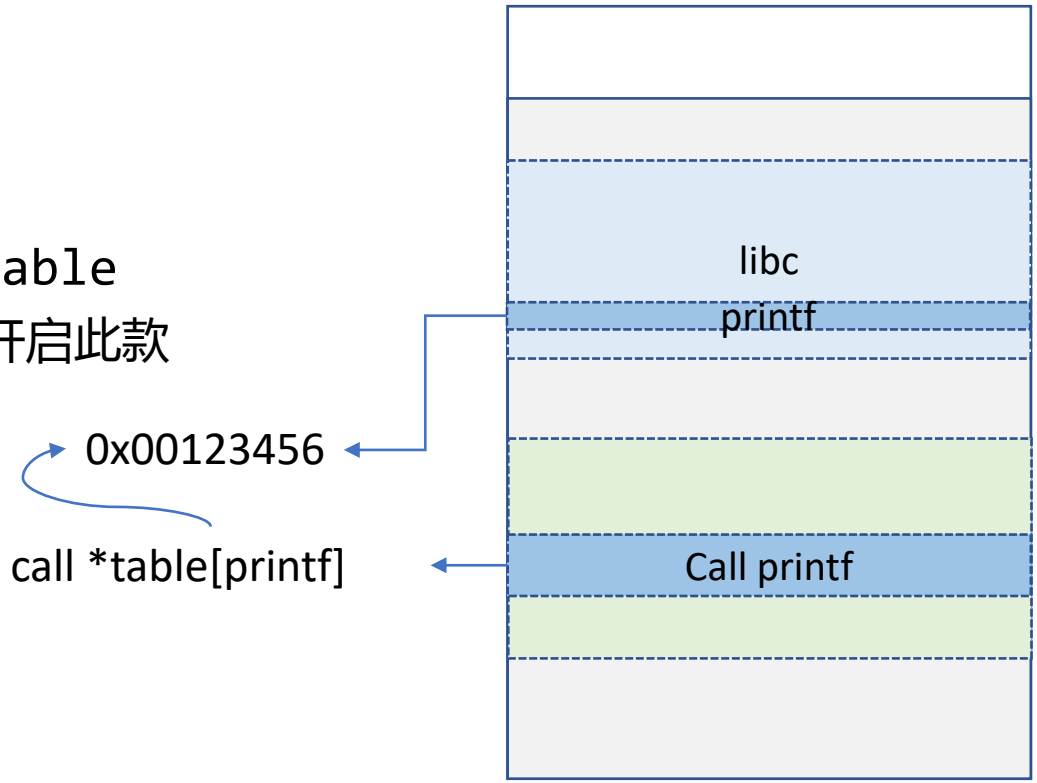


ELF：查表

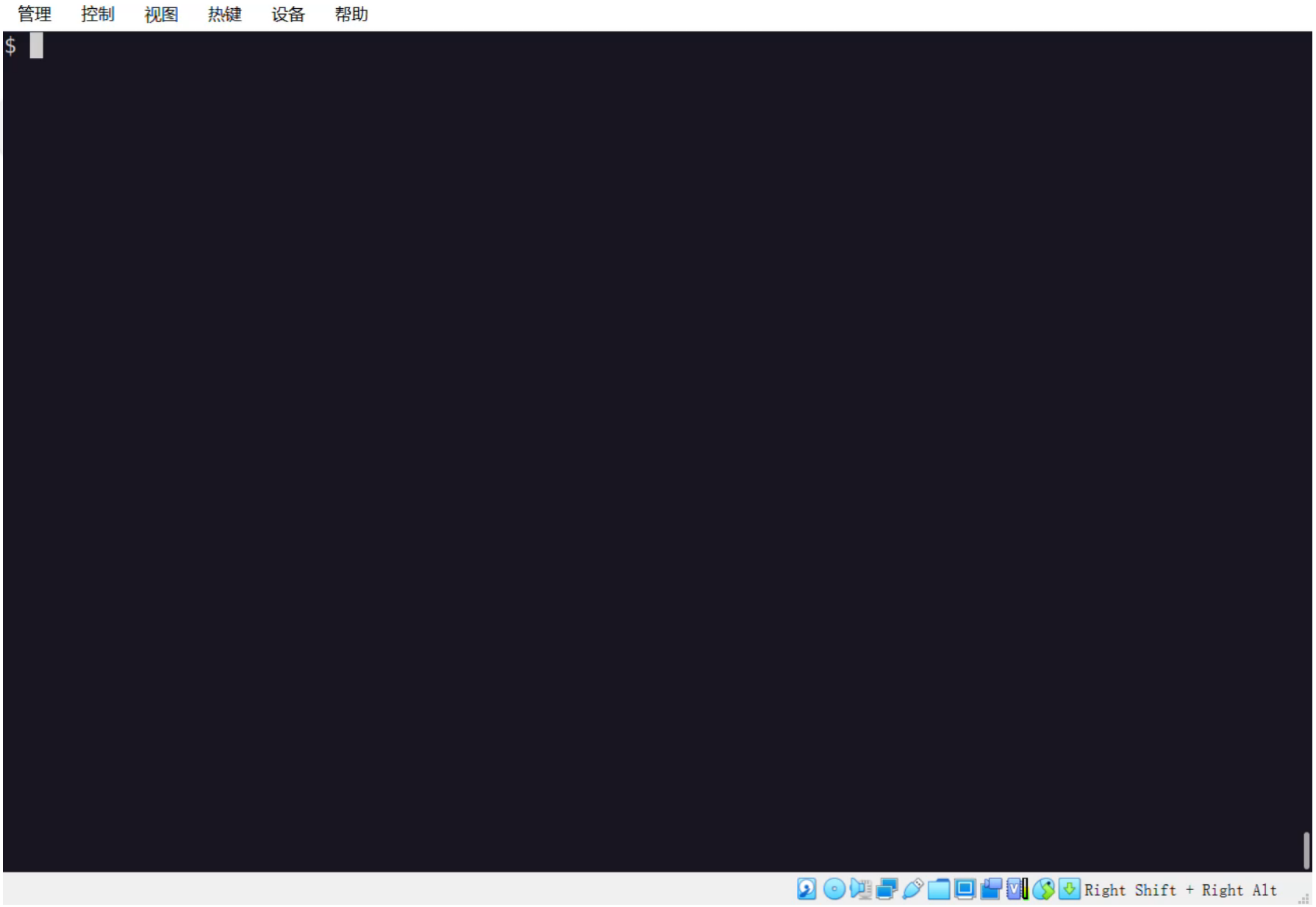
- 基本款

```
call *table[PRINTF]
```

  - 在链接时，填入运行时的table
    - 使用-fno-plt选项可以开启此款



ELF：查表



ELF：查表

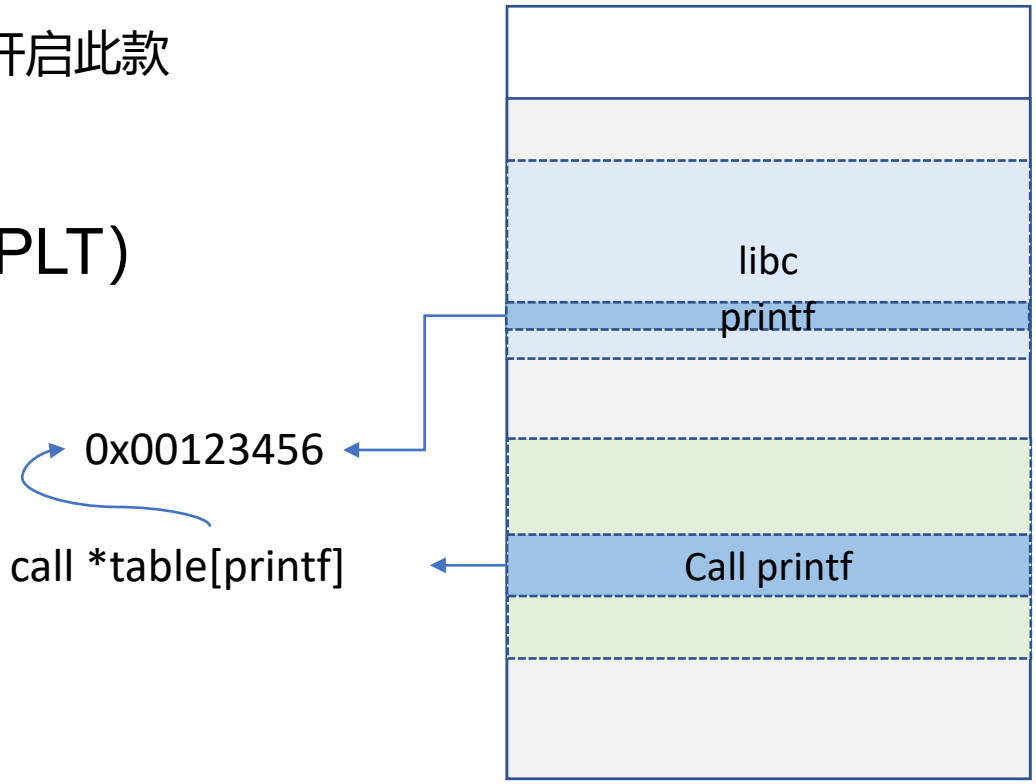
- 基本款

```
call *table[PRINTF]
```

  - 在链接时，填入运行时的table
    - 使用-fno-plt选项可以开启此款

- 豪华款（默认选项，使用PLT）

```
printf@plt:
    jmp *table[PRINTF]
    push $PRINTF
    call resolve
```



总结和反思

为什么我从来没听说过这些知识？？？

因为百度搜不到啊？

- 中文社区几乎不存在这些知识的详细解释
  - the friendly manual是英文写的
  - mailing list/StackOverflow都是英文
  - 国内的专家本来就很少
    - 仅有的那些也没空写文档
- 你们才是未来的希望
  - 不要动不动就说内卷
  - 不要为了无意义的GPA沾沾自喜
  - 不要停止自我救赎

End.

(RTFM; STFW; RTFSC)

还有一个福利

理论课例子

管理 控制 视图 热键 设备 帮助

```
$ cat Lib.h
/*Lib.h*/
#ifndef LIB_H
#define LIB_H
void foobar(int i);
#endif
$ cat Lib.c
/*Lib.c*/
#include <stdio.h>
void foobar(int i){
                printf("Printing from Lib.so %d\n", i);
                sleep(-1);
}
$ cat Program1.c
/*Porgram1.c*/
#include "Lib.h"
int main(){
                foobar(1);
                return 0;
}
$ cat Program2.c
/*Program2.c*/
#include "Lib.h"
int main(){
                foobar(2);
                return 0;
}
$
```


Right Shift + Ri

动态链接.got.plt

管理 控制 视图 热键 设备 帮助

```
C/Lib.so
7f7fd02fe000-7f7fd02ff000 r-xp 00001000 08:03 9444077                    /home/why/Documents/ICS2021/teach/PI
C/Lib.so
7f7fd02ff000-7f7fd0300000 r--p 00002000 08:03 9444077                    /home/why/Documents/ICS2021/teach/PI
C/Lib.so
7f7fd0300000-7f7fd0301000 r--p 00002000 08:03 9444077                    /home/why/Documents/ICS2021/teach/PI
C/Lib.so
7f7fd0301000-7f7fd0302000 rw-p 00003000 08:03 9444077                    /home/why/Documents/ICS2021/teach/PI
C/Lib.so
7f7fd0302000-7f7fd0304000 rw-p 00000000 00:00 0
7f7fd0304000-7f7fd0305000 r--p 00000000 08:03 4726417                    /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f7fd0305000-7f7fd032c000 r-xp 00001000 08:03 4726417                    /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f7fd032c000-7f7fd0336000 r--p 00028000 08:03 4726417                    /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f7fd0336000-7f7fd0338000 r--p 00031000 08:03 4726417                    /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f7fd0338000-7f7fd033a000 rw-p 00033000 08:03 4726417                    /usr/lib/x86_64-linux-gnu/ld-2.33.so
7fff71e8a000-7fff71eab000 rw-p 00000000 00:00 0                          [stack]
7fff71fdf000-7fff71fe3000 r--p 00000000 00:00 0                          [vvar]
7fff71fe3000-7fff71fe5000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
$ readelf -sD Lib.so

Symbol table for image contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5 (2)
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     7: 0000000000001139    55 FUNC    GLOBAL DEFAULT   14 foobar
$
```

Right Shift + Right Alt