链接器、加载器
Linker and Loader

董渊
dongyuan@tsinghua.edu.cn

清华大学
计算机科学与技术系
软件研究所

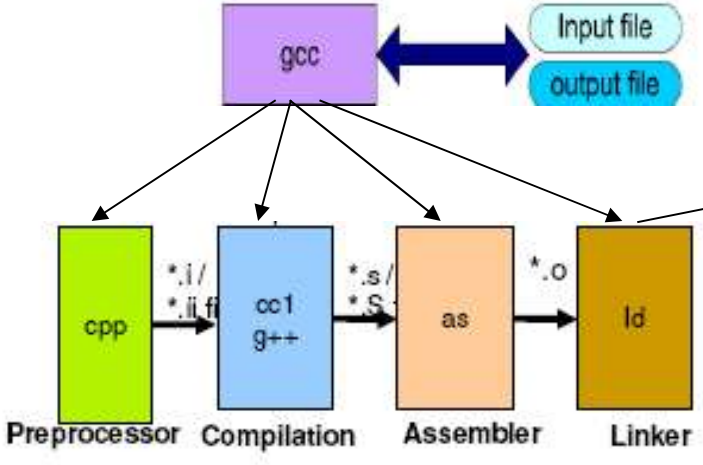
提纲

- 链接和加载的作用
- 静态链接
  - 目标文件 (处理目标)
  - 可重定位目标文件（输入）
  - 符号表和符号解析
  - 重定位
  - 可执行目标文件（输出）
- 加载可执行目标文件
- 共享库的加载和动态链接
- 处理目标文件的常用工具

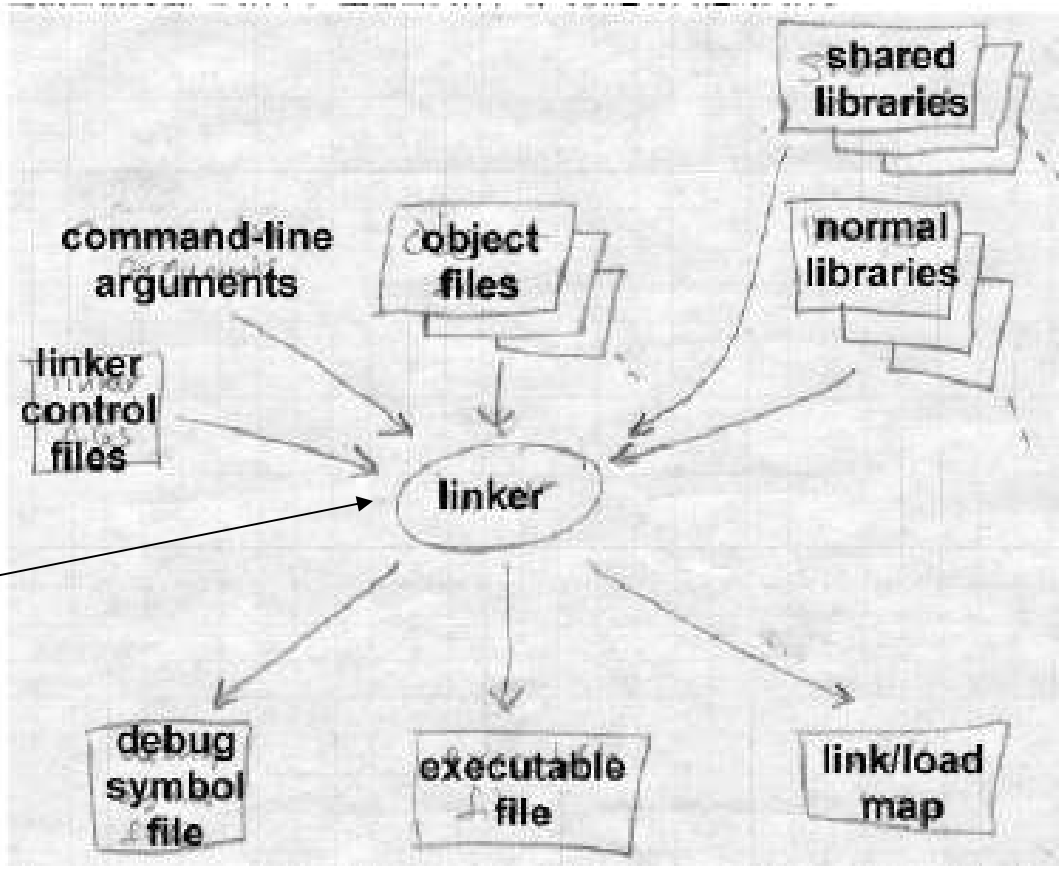
链接和加载的作用

链接器（Linker）是把不同部分的代码和数据，收集、组合成为一个可加载、可执行的文件。

加载器（Loader）把可执行文件从外存装入内存并进行执行



How the GCC works ?



链接和加载的作用

链接的时机：

- 编译时，就是源代码被编译成机器代码时；
- 加载时，也就是程序被加载到内存时；
- 运行时，由应用程序来实施。

链接在现代软件开发中占有极为重要的地位：
为“复用”提供技术支持：

- 它使得分离编译成为可能。我们可以把大型的应用程序分解为多个较小的、便于管理的模块，独立的修改和编译这些模块，这样当我们修改这些模块中的一个时，只需要重新编译链接它，而不必重新编译整个应用程序所有模块。
- 动态绑定（ Binding ）：接口，定义、实现、使用的分离

学习链接知识的意义：

- 帮助我们构造大型程序；
- 帮助我们避免一些危险的编程错误；
- 帮助我们开发共享库（构件库）；

编译器驱动程序

- 编译器驱动程序为用户根据需求调用预处理器、汇编器和链接器。
- 以 GCC 为例，我们要用用 GCC 编译系统编译如下程序：

```
/*    main.c */
void swap();

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

int main()
{
    swap();
    return 0;
}
```

```
/*    swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;
void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

编译器驱动程序

需要在 shell 中输入如下命令来调用 GCC 驱动程序：

unix > gcc –O2 –g –o p main.c swap.c

该命令调用了 GCC 编译器驱动程序，将应用程序从 ASCII 码源文件翻译成可执行的目标文件。在这一过程中，经过了如下步骤：

1. 调用 C 预处理器把 main.c 翻译成中间文件 main.i ；

   cpp [other arguments] main.c /tmp/main.i

2. 调用 C 编译器把 main.i 翻译成汇编语言文件 main.s ；

   cc /tmp/main.i main.c –O2 [other arguments] –o /tmp/main.s

3. 调用汇编器把 main.s 翻译成可重定位目标文件 main.o ；

   as [other arguments] –o /tmp/main.o /tmp/main.s

4. 对 swap.c 执行相同的步骤生成 swap.o ；

5. 调用链接器，将 main.o 和 swap.o 以及必要的系统目标文件链接组合，生成一个可执行目标文件 p 。

   ld –o p [system object files and args] /tmp/main.o /tmp/swap.o

静态链接

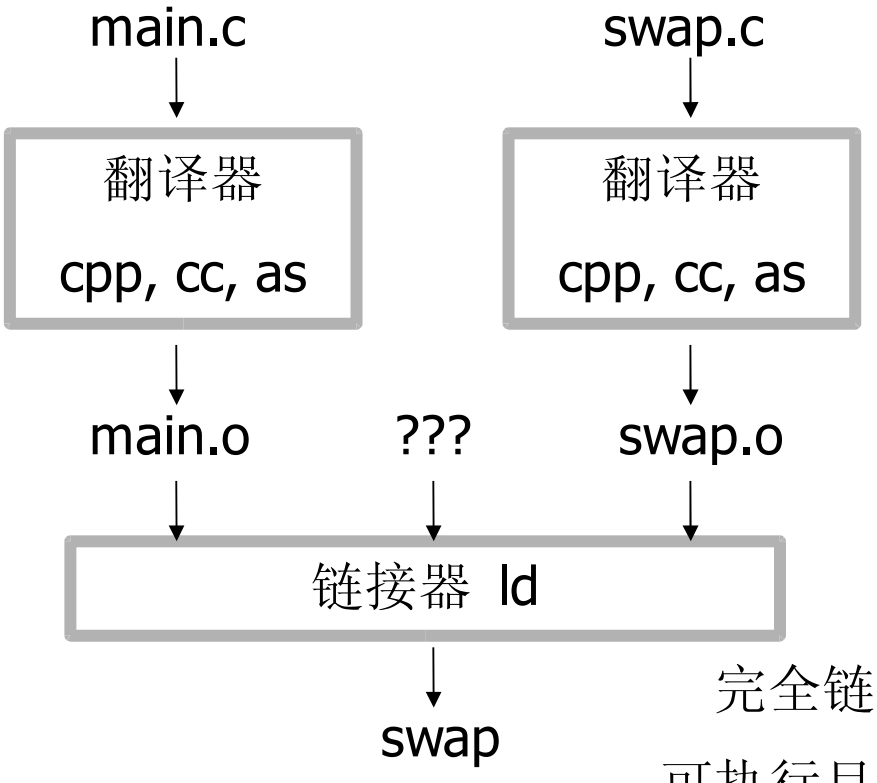
静态连接器是以一组可重定位目标文件和命令行参数作为输入，生成一个完全链接的可以加载和运行的可执行目标文件作为输出。

为创建可执行文件，链接器必须完成主要任务：

- 符号解析：把目标文件中符号的定义和引用联系起来。
- 重定位：把符号定义和内存地址对应起来，然后修改所有对符号的引用。

静态链接

示例程序的静态链接过程描述如下：



目标文件

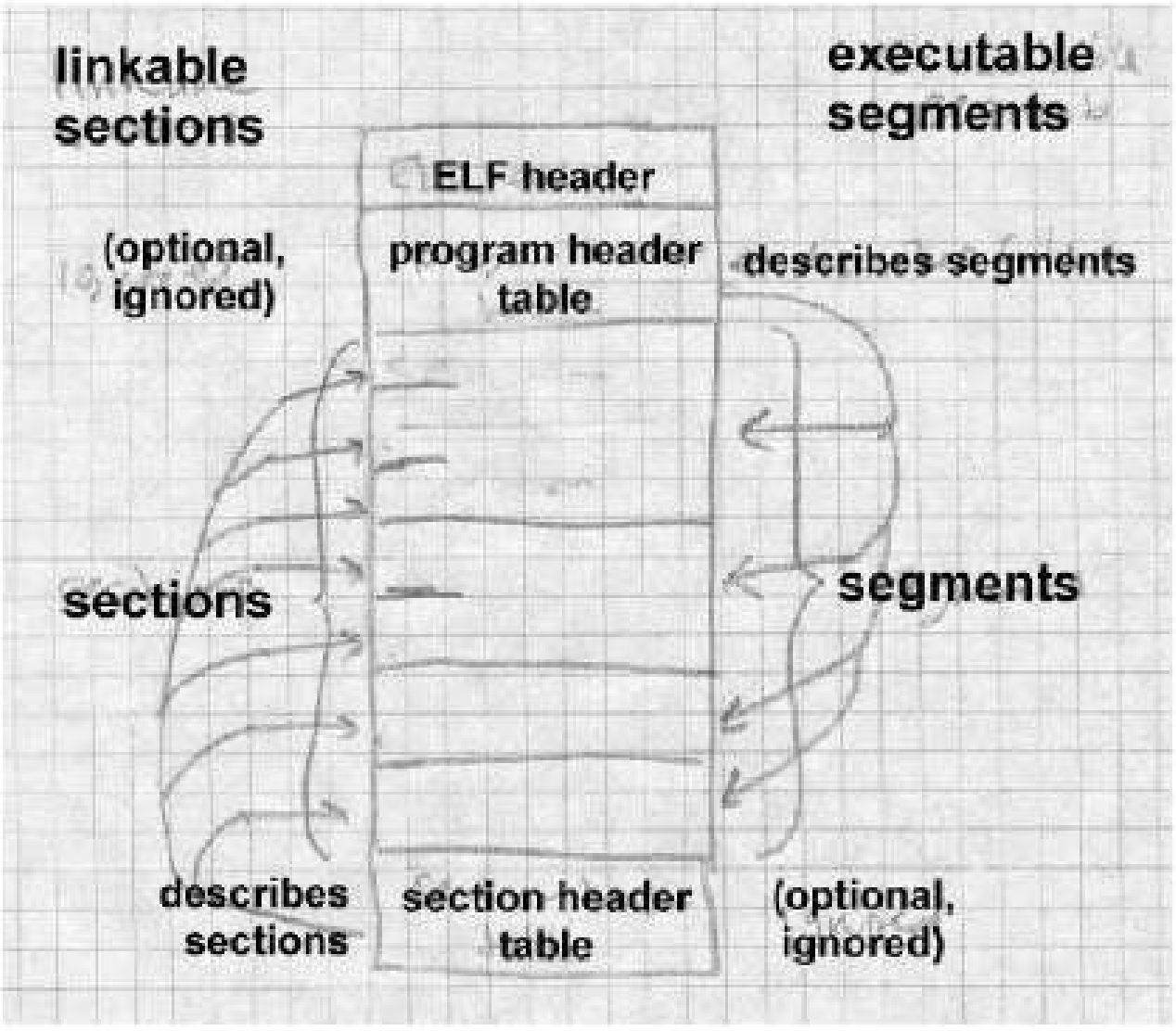
目标文件（Object Files）有三种形式：

- 可重定位（Relocatable）目标文件：由编译器和汇编器生成，可以与其他可重定位目标文件合并创建一个可执行目标文件；
- 可执行（Executable）目标文件：由链接器生成，可以直接复制到内存中执行；
- 共享（shared）目标文件：一类特殊的可重定位目标文件，可以在加载时或运行时被动态的加载到内存并执行。

目标文件

目标文件由各种不同的代码节（code section）和数据节（data section）组成。例如编译产生的机器代码保存在 .text 节中，初始化的全局变量保存在 .data 节中，未初始化的全局变量保存在 .bss 节中。

现代 UNIX 系统，比如 Linux 、 Solaris ，使用 Unix ELF （Executable and Linkable Format）作为目标文件的格式。早期的目标文件格式还包括 a.out 和 COFF （Common Object File Format）格式， WINDOWS 采用的格式是 COFF 的一个变种——PE （Portable and Executable）格式。


linkable sections
executable segments
ELF header
(optional, ignored)
program header table
describes segments
sections
segments
describes sections
section header table
(optional, ignored)

可重定位目标文件

- 典型 ELF 可重定位目标文件的结构

ELF 头
.text （已编译的机器代码）
.rodata （只读数据）
.data （已初始化的全局变量）
.bss （未初始化的全局变量）
.symtab （符号表）
.rel.text （.text 节中需要修改的位置）
.rel.data （本模块中引用或定义的全局变量的可重定位信息）
.debug* （调试符号表，包括局部变量定义）
.line （源程序中行号和 .text 节中指令的映射）
.strtab(字符串表，以 NULL 为结尾的字符串序列)
section header table

夹在 ELF 头和节头部表之间的都是节， ELF 头最开始的 16 个字节描述了字的大小以及生成该文件的系统的字节顺序，剩下的部分都是关于目标文件的信息，包括 ELF 头的大小、目标文件的类型、机器类型、节头部表的偏移量、以及节头部表中表目的大小和数量。

节头部表描述了各个节的位置和大小。

符号表和符号解析

每个可重定位目标模块 m 都有一个符号表，在链接器的上下文中，有三种不同的符号：

- 由 m 定义的，可以被其他模块引用的全局符号，对应于非静态的函数，以及被定义为不带 static 属性的全局变量。
- 由其他模块定义的，被 m 引用的全局符号，对应于定义在其他模块中的函数和变量。
- 只被 m 定义和引用的本地符号。对应于带 static 属性的函数和全局变量，节名、文件名等相关信息。

注：本地符号和本地程序变量（局部变量）是不同的， .symtab 中的符号表不包含本地程序变量的任何符号，这些符号运行时用栈管理。但是带有 static 属性的本地程序变量是例外，编译器在 .data 和 .bss 节中为每个定义分配空间，并在符号表中创建一个有唯一名字的本地符号。

符号表表目数据结构

符号表中表目的数据结构：

```
typedef struct {
        int name;          /*.strtab 节中字符串表的偏移量，指向符号的名字 */
        int value;         /* 距离定义该对象的节的起始位置的偏移 ( 对可重定
                           位目标文件 ) 或绝对运行时地址 ( 对可执行目标文件 )
*/
        int size;          /* 符号代表的对象的大小，单位是字节 */
        char type:4        /* 数据或函数 */
             binding:4     /* 本地或全局 */
        char reserved;     /* 保留，未使用 */
        char section;      /* 该符号相关的节的索引 */
}ELF_Symbol
```

符号表示例

- 下面是 main.o 的符号表的最后 3 个表目：

Num	Value	Size	Type	Bind	Ot	Ndx	Name
7	0	8	OBJECT	GLOBAL	0	3	buf
8	0	28	FUNC	GLOBAL	0	1	main
9	0	0	NOTYPE	GLOBAL	0	UND	swap

- 下面是 swap. o 的符号表的最后 4 个表目：

Num	Value	Size	Type	Bind	Ot	Ndx	Name
7	0	4	OBJECT	GLOBAL	0	3	bufp0
8	0	0	NOTYPE	GLOBAL	0	UND	buf
9	0	54	FUNC	GLOBAL	0	1	swap
10	4	4	OBJECT	GLOBAL	0	COM	bufp1

符号引用的解析

链接器解析符号引用的方法是：将每个引用与输入目标文件的符号表中的一个确定的符号定义联系起来。

对于本地符号（定义和引用均在相同模块中的符号），符号解析简单明了，编译器只允许每个模块中的每个本地符号有一个定义，并确保静态本地变量拥有唯一的本地链接器符号；

全局符号解析

对于全局符号的引用，符号解析要复杂许多，这是由于：

1. 如果该符号不是在当前模块中定义的，编译器假设该符号是在其他模块中被定义，生成一个链接器符号表表目，交给链接器处理，若最终未找到，则输出错误并中止；
2. 此外，相同的全局符号可能会被多个目标模块定义，在这种情况下，链接器将根据一定的原则进行相应的处理。

全局符号解析

在编译时，编译器将每一个全局符号标记为 strong 和 weak 两类：

- 函数和初始化的全局符号被标记为 strong
- 未初始化的全局符号被标记为 weak

链接时，链接器对多重定义的全局符号的解析原则如下：

- 同一个符号不允许有多个 strong 定义；
- 假如一个符号有一个 strong 定义和多个 weak 定义，那么采用该符号的 strong 定义；
- 假如一个符号有多个 weak 定义，那么选取任意一个 weak 定义；

与静态库链接

链接器除了从命令行读取一组可重定位目标文件，把它们链接成为一个可执行目标文件，还提供另外一种机制，就是把静态库作为链接器的输入，构造可执行文件。

静态库（ static library ）就是将相关的目标模块打包形成的单独的文件，使用静态库的优点在于：

- 程序员不需要显式的指定所有需要链接的目标模块，因为指定是一个耗时且容易出错的过程；
- 链接时，连接程序只从静态库中拷贝被程序引用的目标模块，这样就减小了可执行文件在磁盘和内存中的大小。

如何创建静态库

假设有两个文件 addvec.c 和 mulvec.c 分别完成向量的加法和乘法，定义它们的头文件为 vector.h ，编译生成的目标文件分别为 addvec.o 和 mulvec.o ，使用如下命令：

unix> ar rcs libv.a addvec.o mulvec.o

就可以生成包含向量加法和乘法模块的静态库 libv.a 。

静态库使用示例

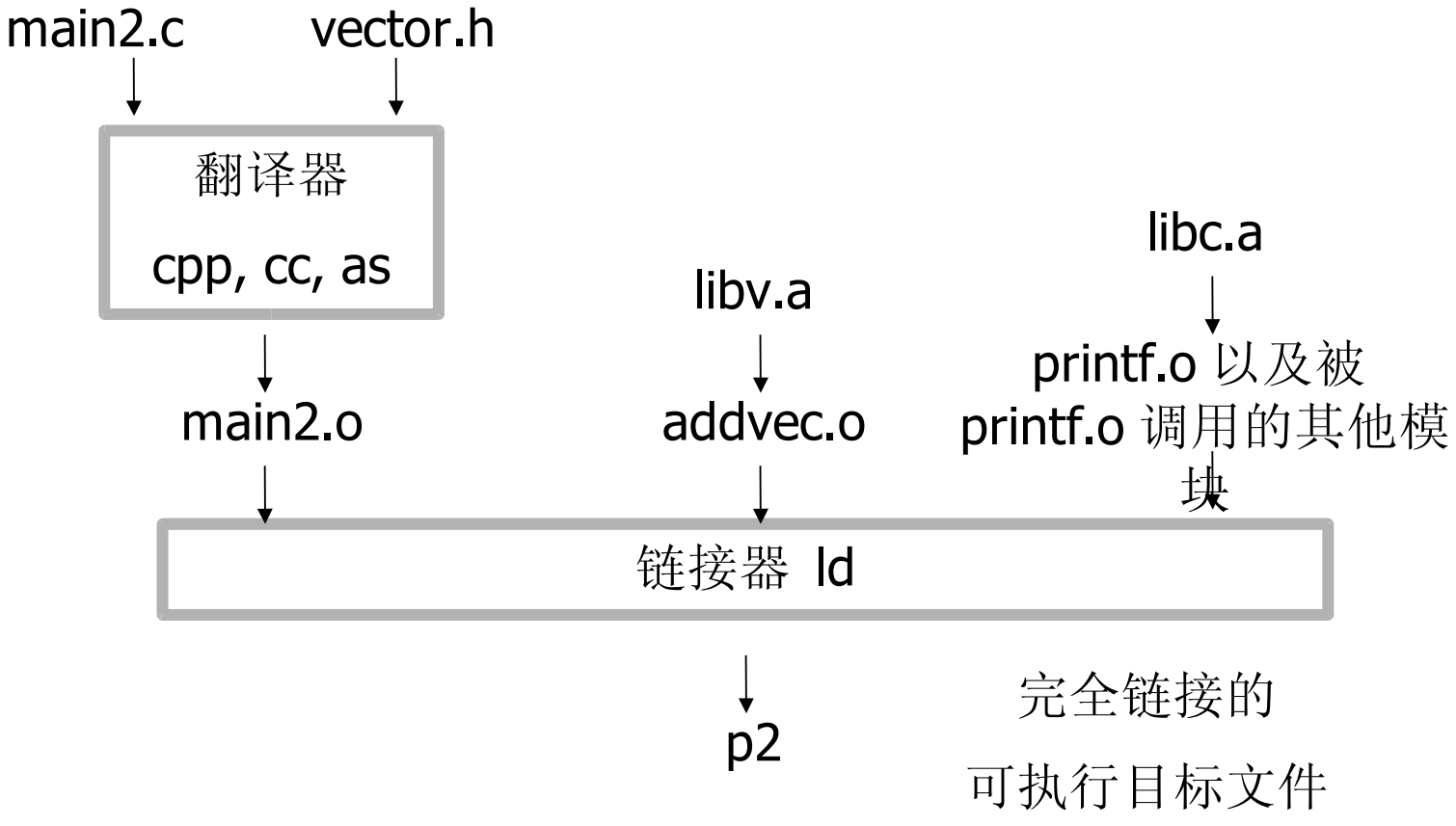
对于如下程序 main2.c，它调用了静态库 libv.a 中向量加法模块 addvec.o 中的函数，因此编译和链接 main2.c 时，需要在命令行中输入如下的参数：

```
/*              main2.c     */
#include <stdio.h>
#include “vector.h”
int x[2] = {1,2};
int y[2] = {3,4};
int z[2];
int main()
{
    addVec(x,y,z,2);
    printf(“z = [%d %d]\n”,z[0],z[1]);
    return 0;
}
```

```
unix> gcc –O2 –c main2.c
```

```
unix> gcc –static –o p2
        main2.o ./libv.a
```

示例程序的静态链接示意图



链接器如何使用静态库解析符号引用

根据从命令行输入的编译器驱动程序命令，链接器从左至右的扫描可重定位目标文件以及库文件。在扫描的过程中，链接器维护着 3 个集合：

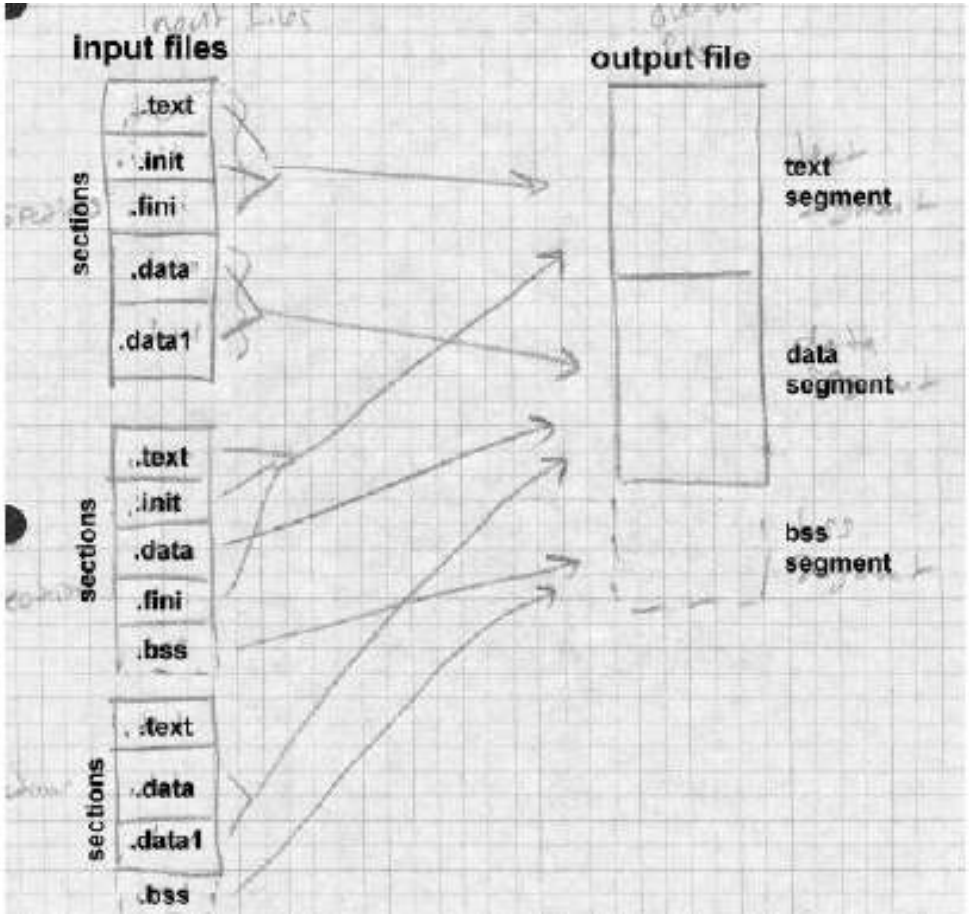
- 即将被合并到可执行目标文件中的目标文件的集合 E；
- 没有解析的符号集合 U；
- 在先前输入的文件中已有定义的符号集合 D 。

链接器如何使用静态库解析符号引用

初始情况下，U、D、E都是空集。然后依据如下算法，依次处理输入文件：

- 对每一个输入文件f，链接器判断f是一个目标文件还是一个库文件。如果f是一个目标文件，把f加入E，根据f中的符号定义和符号引用更新U和D，然后处理下一个文件。
- 如果f是库文件，链接器尝试在库文件的成员中寻找U中符号（还没有解析的符号）的定义。如果某个成员m中包含U中某个符号的定义，那么把m加入E，根据m中的符号定义和符号引用更新U和D。重复这一过程直到U和D都不再变化，然后把不在E中的库文件的成员直接丢弃；
- 当链接器扫描完所有输入文件后，如果U是非空的，则报错并退出；如果U已经是空集，则合并E中的所有目标文件，形成可执行文件。

目标文件合并



Link scripts，链接脚本

重定位

在链接器完成了符号解析工作之后，每一个符号引用都与一个确定的符号定义相关联起来，链接器也知道了每一个输入模块的代码节和数据节的大小，然后开始重定位过程。

在重定位过程中，链接器将合并所有输入模块，为每个符号分配运行时地址。由两个步骤组成：

- 重定位节和符号定义。首先将所有输入文件中类型相同的节合并成新的聚合节，然后把运行时内存地址赋给聚合节、输入模块中的每一个节、以及输入模块中定义的每一个符号。
- 重定位节中的符号引用。链接器修改代码节和数据节中对每一个符号的引用，使它们指向正确的运行时地址，这一步骤依赖于重定位表目（relocation entry）。

重定位表目

当汇编器遇到最终位置未知的目标引用，就会生成一个重定位表目，告诉链接器如何修改这个引用，代码的重定位表目放在 .rel.text 节中，已初始化数据的重定位表目放在 .rel.data 节中， ELF 重定位表目的数据结构如下：

```
typedef struct{
        int offset;        /* 需要重定位的符号引用的节偏移地址 */
        int symbol:24;  /* 被修改引用应该指向的符号 */
            type:8;       /* 重定位类型，告诉链接器如何修改新引用
*/
}Elf32_Rel;
```

两种基本的重定位类型

R_386_PC32：重定位一个使用 32 位 PC 的相关地址引用；

R_386_32：重定位一个使用 32 位绝对地址的引用；

重定位符号引用算法

```
foreach section s{
        foreach relocation entry r{
          refptr = s + r.offset;
          if ( r.type == R_386_PC32 )    {
           refaddr = ADDR(s) + r.offset;
           *refptr = (unsigned) ( ADDR(r.symbol)
                    +*refptr-refaddr);
          }
          if ( r.type == R_386_32 )
           *refptr = (unsigned) ( ADDR(r.symbol)
                    +*refptr);
        }
      }
```

重定位符号示例

在 7.2 所示的程序中， main.o 调用了 swap.o 中定义的函数 swap ，call 指令的反汇编结果如下：

6: e8 fc ff ff ff call 7<main+0x7> *swap();*

7: R_386_PC32 swap *relocation entry*

其中 relocation entry 含义为：

r.offset = 0x7; r.symbol = swap; r.type = R_386_PC32

假设链接器已经确定：

ADDR(s) = ADDR(.text) =0x80483b4

ADDR(r.symbol) = ADDR(swap) = 0x80483c8

则重定位计算如下：

refaddr = ADDR(s) + r.offset = 0x80483bb

refptr = (unsigned) (ADDR(r.symbol) +*refptr - refaddr)

= (unsigned) (0x80483c8 + (-4) – 0x80483bb)

= 0x9

重定位符号示例

最终，经过修改的可执行文件中 call 指令所在行的内容如下：

```
80483ba: e8 09 00 00 00  call 80483c8 <swap>
```

注意，在上面的计算过程中， *refptr 的初始值之所以为 -4 ，这是由于 PC 总是指向当前指令的下一条指令，汇编器为了表示这一事实，把 -4 作为缺省值。在不同指令长度和编码方式的其他体系结构中，这一值也可能不同。

可执行目标文件

- 典型 ELF 可执行目标文件的结构

ELF 头
segment header table
.init （初始化代码）
.text （已编译的机器代码）
.rodata （只读数据）
.data （已初始化的全局变量）
.bss （未初始化的全局变量）
.symtab （符号表）
.debug （调试符号表，包括局部变量定义）
.line （源程序中行号和 .text 节中指令的映射）
.strtab(字符串表，以 NULL 为结尾的字符串序列)
section header table

与可重定位目标文件的格式相似，不同之处在于：

- ELF 头还包含程序的入口点，也就是程序要执行的第一条指令的地址。
- 不再包含 .rel 节，包含描述初始化信息的 .init 节，包含描述组块和内存映射关系的段头部表（ segmen header table ）。

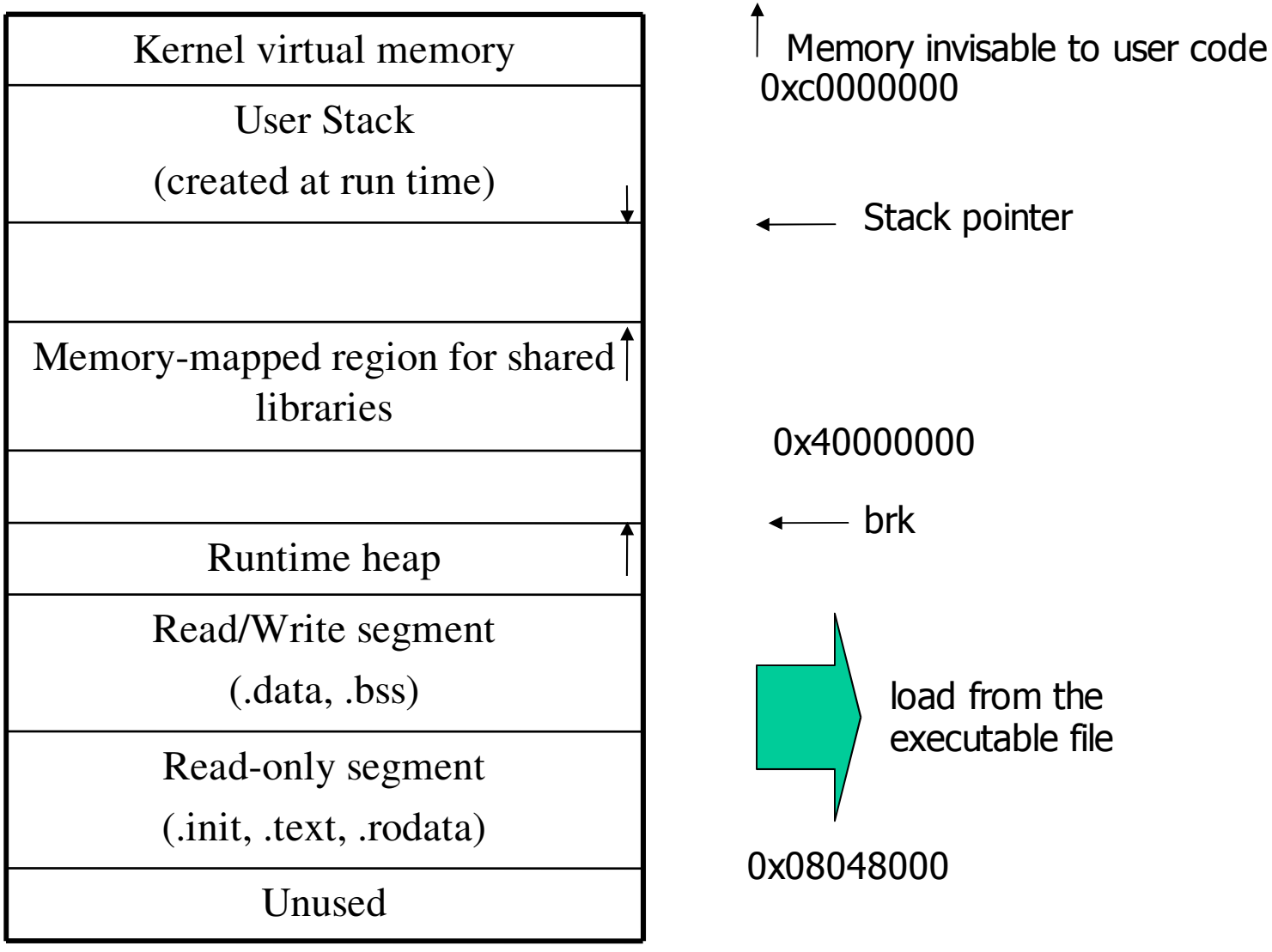
加载可执行目标文件

在命令行输入如下命令就能够运行可执行文件 p：

unix> ./p

在 UNIX 系统中， shell 假设 p 是一个可执行目标文件，调用常驻内存的加载器（ *loader* ）程序，把 p 中的代码和数据拷贝到内存中，然后跳转到 p 的第一条指令，执行 p 。这个将程序拷贝到内存并运行它的过程就被称作是加载（ *loading* ）。任何一个程序都可以通过调用 *execve* 函数来调用 *loader* 。

Linux 的运行时内存映像



加载器的工作过程

加载器首先创建如图所示的内存映像，然后根据段头部表，把目标文件拷贝到内存的数据和代码段中。然后，加载器跳转到程序入口点（即符号 _start 的地址），执行启动代码（ startup code ），启动代码的调用顺序如所示：

```
1 0x080480c0  <_start>:
2   call __libc_init_first
3   call _init
4   call atexit
5   call main
6   call _exit
```

在执行完初始化任务后，启动代码调用 atexit 例程，该例程注册了一系列调用 exit 函数时必须的例程；然后启动代码调用应用程序的 main 例程，执行用户程序代码；当用户程序返回后，启动代码调用 _exit 例程，将控制权交还给操作系统。

函数调用关系和运行时库（ glibc ）的版本有关

共享库的加载和动态链接

共享库是基于静态库的不足而产生的，这些不足包括：

- 静态库需要定期维护和更新，如果程序员需要自己的程序使用静态库的最新版本，那他首先要保持关注静态库的最新变化，并且在静态库变化之后对程序进行重新链接；

- 几乎每个 C 程序都包括很多相同的 I/O 函数，因此在运行时，每一个程序的代码节中都包含这些重复的函数，这就严重的浪费了内存资源。

共享库特征

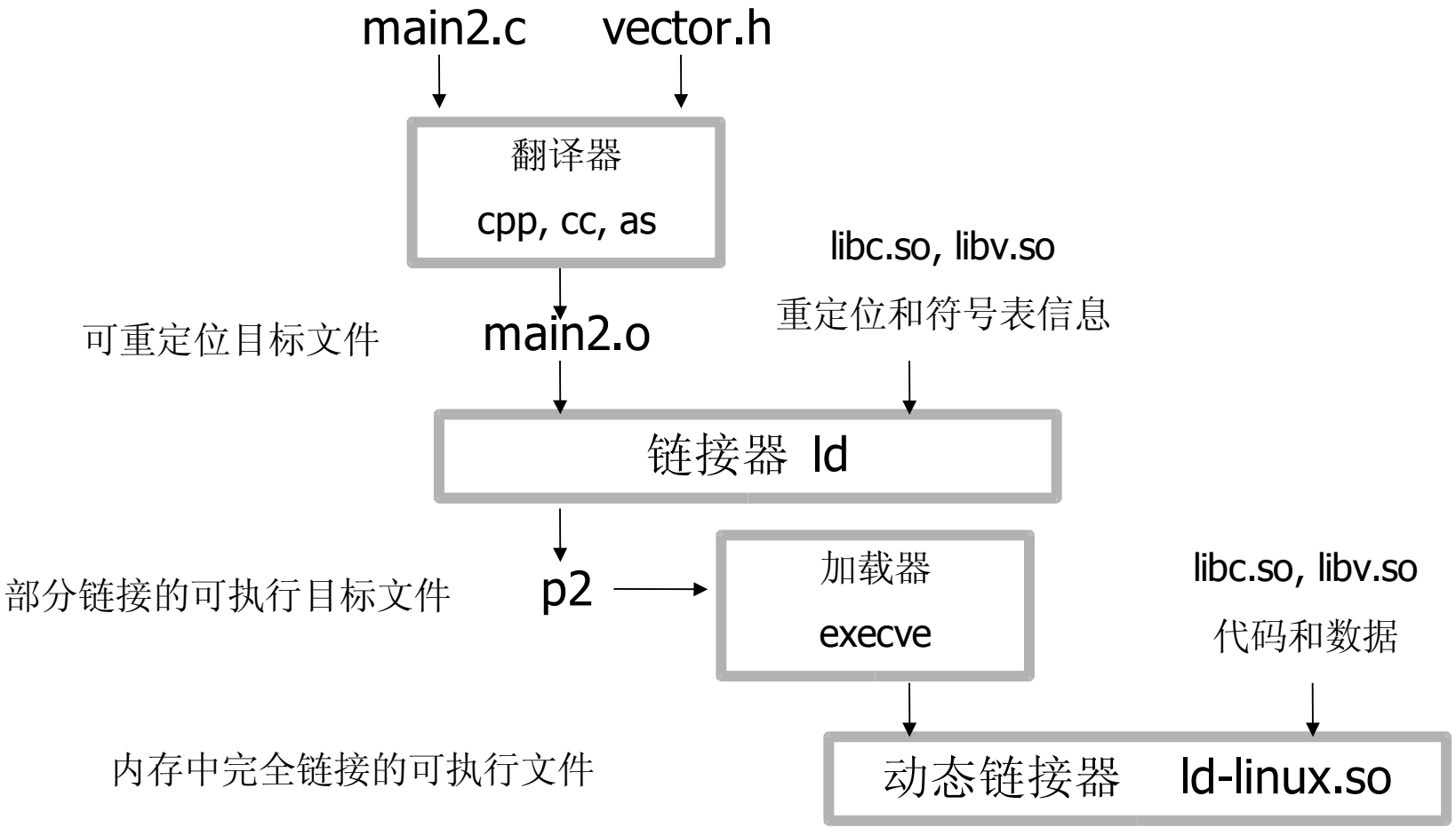
共享库解决了静态库的不足，共享库是一种特殊的目标模块，它可以在运行时被加载到任意的内存地址，或者是与任意的程序进行链接。

这个过程被成为动态链接，而完成这一功能的程序被称为动态链接器。

共享库在 UNIX 系统中通常后缀为 .so ，而在 WINDOWS 中，共享库是指 DLLs(dynamic link library) 。

动态链接共享库

下图以图 7.6 中向量运算的程序为例，概括了程序的动态链接过程：



动态链接共享库

在上面所示的例子中， p2 没有拷贝任何 libv 的代码节和数据节，仅拷贝了一些重定位和符号表信息，这样就可以在运行时解析对 libv.so 中代码和数据的引用。

当加载器加载和运行 p2 时，注意到 p2 中包含一个 .interp 节，该节包含动态链接器的路径名，于是加载器加载和运行这个动态链接器 (动态链接器本身也是一个共享目标) 。

动态链接器通过执行下面重定位完成链接：

- 重定位 libc.so 的代码和数据到某个内存段；
- 重定位 libv.so 的代码和数据到另一个内存段；
- 重定位 p2 中所有由 libc.so, libv.so 定义的符号和引用。

应用程序中加载和链接共享库

除了在编译时、加载时完成对共享库的链接，有时候还需要在运行时由动态链接器加载和链接共享库。

UNIX 系统为动态链接器提供了一个简单的接口，使应用程序能够在运行时加载和链接共享库：

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, char *symbol);
void *dlclose(void *handle);
void *dlerror(void);
```

处理目标文件的常用工具

UNIX 系统提供了一系列工具帮助理解和处理目标文件。 GNU binutils 包也提供了很多帮助。这些工具包括：

- AR ： 创建静态库，插入、删除、列出和提取成员；
- STRINGS ： 列出目标文件中所有可以打印的字符串；
- STRIP ： 从目标文件中删除符号表信息；
- NM ： 列出目标文件符号表中定义的符号；
- SIZE ： 列出目标文件中节的名字和大小；
- READELF ： 显示一个目标文件的完整结构，包括 ELF 头中编码的所有信息。
- OBJDUMP ： 显示目标文件的所有信息，最有用的功能是反汇编 .text 节中的二进制指令。
- LDD ： 列出可执行文件在运行时需要的共享库。