第8章 编译系统和运行系统

本章内容

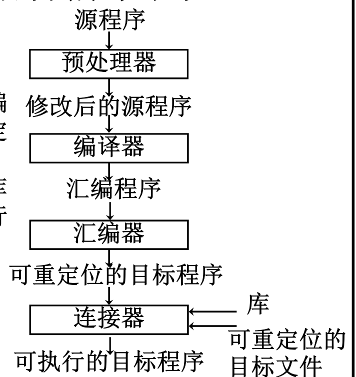
- C语言编译系统
  - 预处理器、编译器、汇编器、连接器
  - 目标文件的格式、静态库、动态连接
- Java运行系统

引入本章的目的

- 掌握从源程序到可执行目标程序的实际处理过程
- 对实际参与软件开发是直接有用的

8.1 C语言编译系统

- C源程序可以分成若干个模块
- 分别进行预处理、编译和汇编、形成可重定位的目标文件
- 目标文件和必要的库文件连接成一个可执行的目标文件
- gcc和cc是编译驱动程序的名字



8.1 C语言编译系统

main.c

```
(1) #if 1
(2) int buf[2];
(3) #else
(4) int buf[2] = {10,20};
(5) #endif
(6) void swap();
(7) #define A buf[0]
(8) int main( )
(9) {
(10)scanf("%d, %d", buf, buf+1);
(11)swap( );
(12)printf("%d, %d",A, buf[1]);
(13)return 0;
(14) }
```

swap.c

```
(1) extern int buf[2];
(2) int *bufp0 = buf;
(3) int *bufp1;
(4) void swap( )
(5) {
(6) int temp;
(7)bufp1 = buf+1;
(8)temp = *bufp0;
(9)*bufp0 = *bufp1;
(10)*bufp1 = temp;
(11) }
```

8.1 C语言编译系统

8.1.1 预处理器

- gcc首先调用预处理器cpp，将源程序文件翻译成一个ASCII中间文件，它是经修改后的源程序
- cpp实现以下功能
  - 文件包含
  - 宏展开
  - 条件编译

8.1 C语言编译系统

main.c

```
(1) #if 1
(2) int buf[2];
(3) #else
(4) int buf[2] = {10,20};
(5) #endif
(6) void swap( );
(7) #define A buf[0]
(8) int main( )
(9) {
(10)scanf("%d, %d", buf, buf+1);
(11)swap( );
(12)printf("%d, %d",A, buf[1]);
(13)return 0;
(14) }
```

main.i

```
(1) # 1 "main.c"
(2)
(3) int buf[2];
(4)
(5)
(6)
(7) void swap( );
(8)
(9) int main( )
(10) {
(11) scanf("%d, %d", buf, buf+1);
(12) swap( );
(13) printf("%d,%d",buf[0], …);
(14) return 0;
(15) }
```

8.1 C语言编译系统

8.1.2 汇编器

- GCC系统的编译器cc1产生汇编代码
- 最简单的汇编器对输入进行两遍扫描
- 一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的
- 用 **gcc –S main.c**

可以得到汇编文件**main.s**

- 用 **as –o main.o main.s**

可以将**main.s**汇编成可重定位目标文件**main.o**

8.1 C语言编译系统

- 例 一段汇编代码

```
.L2:
        cmpl $0,-4(%ebp)
        jne .L6
        jmp .L11
.L11:
        cmpl $0,-8(%ebp)
        jne .L6
        jmp .L12
.L12:
        jmp .L5
        .p2align 4,,7
.L6:
```

第一编扫描建立符号表，包括代码标号.L2、.L11等

第二遍扫描依据符号表中的信息来产生可重定位代码

8.1 C语言编译系统

8.1.3 连接器

目标模块或目标文件的形式

- 可重定位的目标文件
- 可执行的目标文件
- 共享目标文件
  - 一种特殊的可重定位目标文件
  - 在装入程序或运行程序时，动态地装入到内存并连接

8.1 C语言编译系统

- 连接是一个收集、组织程序所需的不同代码和数据的过程，以便程序能被装入内存并被执行
- 连接的时机
  - 编译时
  - 装入时
  - 运行时
- 静态连接器
- 动态连接器

8.1 C语言编译系统

- 一个重定位模块**M**可能定义和引用的符号
  - 全局符号 指那些在模块**M**中定义，可以被其他模块引用的符号
  - 局部符号 指那些在模块**M**中定义，且只能在本模块中引用的符号
  - 外部符号 指那些由模块**M**引用并由其他模块定义的符号
- 符号解析
  - 识别各个目标模块中定义和引用的符号，为每一个符号引用确定它所关联的一个同名符号的定义
- 重定位

8.1 C语言编译系统

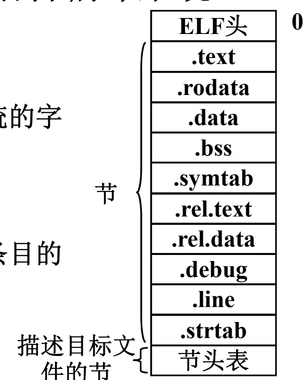
8.1.4 目标文件的格式

- 目标文件格式随系统不同而不同
- 介绍**Unix**的**ELF**（***Executable and Linkable Format***）格式
- **Linux**、**System V Unix**的后期版本、**BSD Unix**变体和**Sun Solaris**，都使用**Unix**的**ELF**格式

8.1 C语言编译系统

ELF头

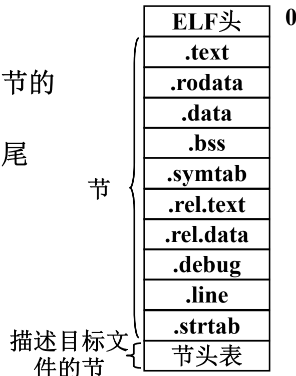
- 描述了字的大小
- 产生此文件的系统的字节次序
- 目标文件的类型
- 机器类型
- 节头表的位置、条目的大小和数量
- 其他



8.1 C语言编译系统

节头表

- 描述目标文件中各节的位置和大小
- 处于目标文件的末尾



8.1 C语言编译系统

.text节

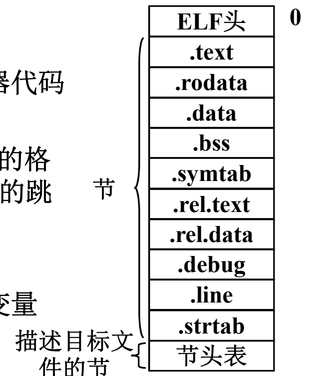
被编译程序的机器代码

.rodata节

诸如printf语句中的格式串和switch语句的跳转表等只读数据

.data节

已初始化的全局变量



8.1 C语言编译系统

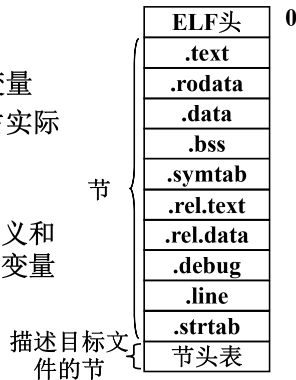
.bss节（.comm 节）

未初始化的全局变量

在目标文件中不占实际的空间

.symtab节

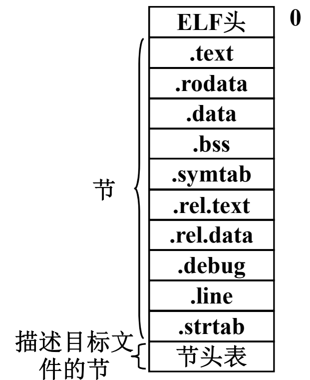
记录在该模块中定义和引用的函数和全局变量的信息的符号表



8.1 C语言编译系统

.symtab节

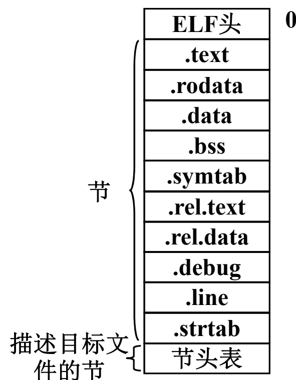
- Type
  - FUNC
  - OBJECT
- Bind
  - GLOBAL
  - LOCAL
  - EXTERN



8.1 C语言编译系统

.symtab节

- Name
- Value
  - 偏移地址，或
  - 绝对地址
- Size
  - 字节数



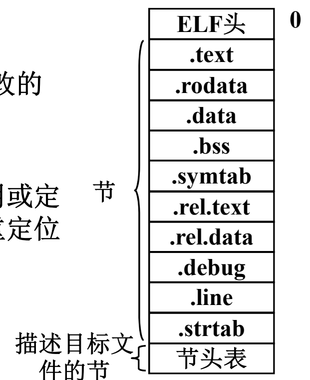
8.1 C语言编译系统

.rel.text节

.text节中需要修改的单元的位置列表

.rel.data节

用于被本模块引用或定义的全局变量的重定位信息



8.1　C语言编译系统

.debug节

用于调试程序的调试符号表

.line节

源文件和**.text**节中的机器指令之间的行号映射

.strtab

一组有空结束符的串构成的串表

ELF头	**0**
.text	节
.rodata	节
.data	节
.bss	节
.symtab	节
.rel.text	节
.rel.data	节
.debug	节
.line	节
.strtab	节
节头表	描述目标文件的节

8.1　C语言编译系统

8.1.5 符号解析

- 将每个符号引用正确地与某可重定位模块的符号表中的一个符号定义相关联，从而确定各个符号引用的位置
- 在所有输入模块中都找不到被引用符号的定义，则打印错误消息并结束连接
- 需要定义解析规则

8.1　C语言编译系统

- 解析规则
  - 函数和已初始化的全局变量称为强符号；未初始化的全局变量称为弱符号
  - 不允许有多重的强符号定义
  - 出现一个强符号定义和多个弱符号定义时，选择强符号的定义
  - 出现多个弱符号定义时，选择任意一个弱符号的定义

8.1　C语言编译系统

8.1.6 静态库

- 将相关的可重定位目标模块打包成一个文件，作为连接器的输入
- 连接器仅复制库中被应用程序引用的模块

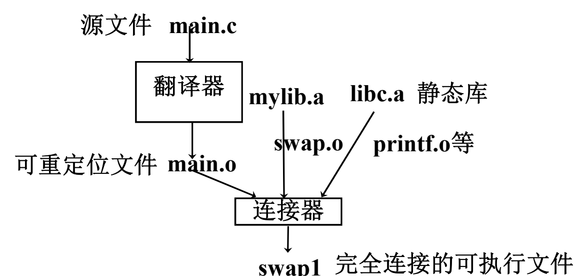
gcc –c swap.c　　——编译

ar rcs mylib.a swap.o　　——建库

gcc　–static　–o　swap1　main.c　/usr/lib/libc.a　mylib.a　　——生成可执行文件

8.1　C语言编译系统

和静态库连接



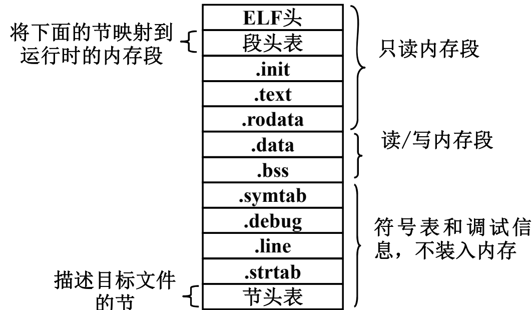
8.1　C语言编译系统

8.1.7 可执行目标文件及装入

- 可执行目标文件与可重定位目标文件格式类似
- 可执行目标文件的装入由加载器完成

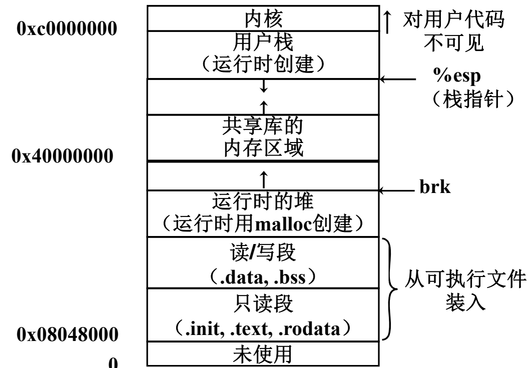
8.1 C语言编译系统

典型的ELF可执行目标文件



8.1 C语言编译系统

Linux运行时的内存映像



8.1 C语言编译系统

- 这里描述的装入过程从概念上来说是正确的
- 若需要了解装入过程真正是怎样工作的，必须在理解了进程、虚拟内存和内存分页等概念以后

8.1 C语言编译系统

8.1.8 动态连接

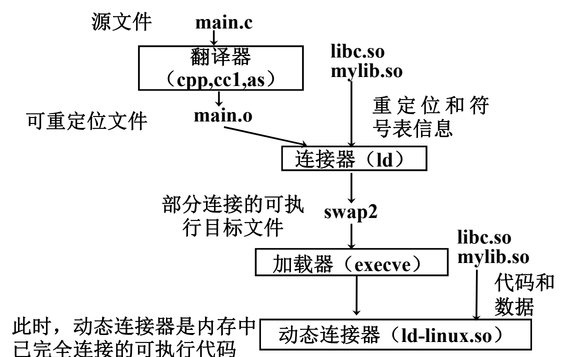
- 静态库
  - 周期性地被维护和更新
  - 内存可能有多份**printf**和**scanf**的代码
- 共享库
  - 在运行时可以装到任意的内存位置，被内存中的进程共享

8.1 C语言编译系统

共享库以两种不同的方式被共享

- 共享库的代码和数据被所有引用该库的可执行目标文件所共享
- 共享库的**.text**节在内存中的一个副本可以被正在运行的不同进程共享

8.1 C语言编译系统



8.1 C语言编译系统

加载器通常装入和运行动态连接器

动态连接器接着完成连接任务
- 把**libc.so**的文本和数据装入内存并进行重定位
- 把**mylib.so**的文本和数据装入内存并进行重定位
- 重定位**swap2**中任何对**libc.so**或**mylib.so**定义的符号的引用
- 将控制传递给应用程序

8.1 C语言编译系统

8.1.9 处理目标文件的一些工具

ar 创建静态库，插入、删除、罗列和提取成员
strings 列出包含在目标文件中的所有可打印串
strip 从一个目标文件中删除符号表信息
nm 列出一个目标文件的符号表中定义的符号
size 列出目标文件中各段的名字和大小
readelf 显示目标文件的完整结构，包括编码在**ELF**头中的所有信息。它包括了**size**和**nm**的功能
objdump 可以显示目标文件中的所有信息。其最有用的功能是反汇编**.text**节中的二进制指令
ldd 列出可执行目标文件在运行时需要的共享库

8.2 Java语言的运行系统

- **Java**语言
  - 简单性、分布性、安全性、可移植性等
  - 关心：平台无关性
- **Java**虚拟机技术是实现**Java**平台无关性特点的关键
- **Java**运行系统就是**Java**虚拟机的一个实现

8.2 Java语言的运行系统

8.2.1 Java虚拟机语言简介

Java程序首先由**Java**编译器把它编译成字节码，也就是**JVML**程序
- 常量池：存放各种字符串常量，类似于传统程序设计语言中的符号表
- 类成员信息：域信息表和方法信息表
- **JVML**指令序列

8.2 Java语言的运行系统

Java源程序中的方法：

```
int calculate (int i){

  int j =2;
  return ((i+j) *(j-1));

}
```

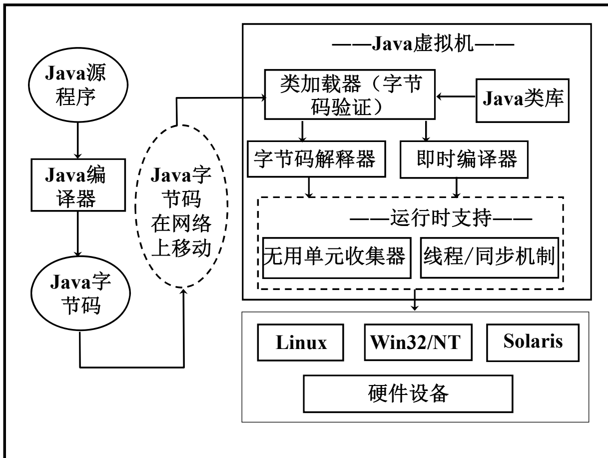
对应的**JVML**指令序列：

```
int calculate (int i)
iconst _2
istore _2
iload _1
iload _2
iadd
iload _2
iconst _1
isub
imul
ireturn
```

8.2 Java语言的运行系统

8.2.2 Java虚拟机

Java虚拟机一般由以下几个部分构成：
- 类加载器（字节码验证器）
- 解释器或/和编译器
- 包括无用单元收集器和线程控制模块在内的运行支持系统
- 另外还有一些标准类和应用接口的**class**文件库



8.2 Java语言的运行系统

- **C**语言
  - 数据栈用来存放生存期和一次过程活动的生存期一致的局部变量
  - 数据堆用来存放生存期和一次过程活动的生存期不一定一致的动态变量
    - 程序员通过**malloc**和**free**函数参与堆的管理，这是 不安全的一个根源（悬空指针、内存泄漏）
- **Java**语言
  - 数据栈除对象和数组外，都分配在栈上
  - 数据堆对象和数组分配在堆上
    - 出于安全的要求，程序员不参与堆管理

8.2 Java语言的运行系统

无用单元收集（俗称垃圾收集）

- 无用单元（理论上）
  - 那些在后续运行过程中不会再使用的数据单元
- 收集器采用稳妥策略
  - 实际上并非总能判断出一个数据记录的值以后是否还需要
  - 通过根集（*roots set*，在栈上）以及从根集开始的可达性来定义对象的活跃性
- 无用单元（实现上）
  - 通常指那些不可能从程序变量经对象引用到达的堆分配记录

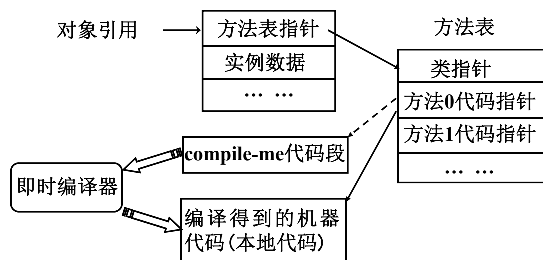
8.2 Java语言的运行系统

8.2.3 即时编译器

- 当一个类的某个方法第一次被调用时，虚拟机才激活即时编译器将它编译成机器代码
- 生成的代码的执行速度可以达到解释执行的**10**倍
- 但是执行过程不得不等待编译的结束，因而使得执行时间变长
- 很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合

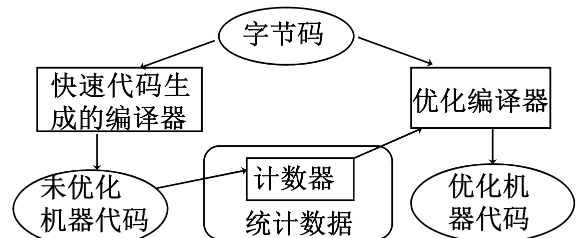
8.2 Java语言的运行系统

即时编译



8.2 Java语言的运行系统

重编译机制



例　题　1

如果cfile是一个C语言源程序（注意，该文件名没有后缀），在X86/Linux机器上，命令

cc cfile

的结果是错误信息

/usr/bin/ld: cfile: file format not recognized: treating as linker script
/usr/bin/ld: cfile: 1: parse error
collect2: ld returned 1 exit status

请解释为什么会是这样的错误信息

例　题　2

下面是C语言的一个程序：

```
long gcd(p,q) long p,q; {
  if (p%q == 0) return q;
  else return gcd(q, p%q);
}
main() {
  printf("\n%ld\n",gcdx(4,12));
}
```

在X86/Linux机器上，用gcc命令得到的编译结果如下

In function 'main':undefined reference to 'gcdx'
ld returned 1 exit status.

请问，这个gcdx没有定义，是在编译时发现的，还是在连接时发现的？试说明理由

例　题　3

一些C程序设计的教材上指出"在需要使用标准I/O库中的函数时，应在程序前使用

#include <stdio.h>

预编译命令，但在用printf和scanf函数时，则可以不要。" 但事实上并非仅限于这两个函数。例如下面的C程序编译后运行时输出字符A并换行，它并没有预编译命令#include <stdio.h>。试解释为什么

```
main()
{
  putchar('A');   putchar('\n');
}
```

例　题　4

C的一个源文件可以包含若干个函数，该源文件经编译可以生成一个目标文件；若干个目标文件可以构成一个函数库

如果一个用户程序引用某函数库中某文件的某个函数，那么，在连接时的做法是下面三种方式的哪一种，说明理由

- 将该函数的目标代码连到用户程序
- 将该函数的目标代码所在的目标文件连到用户程序
- 将该函数库全部连到用户程序

例　题　5

cc是UNIX系统上C语言编译命令，–l是连接库函数的选择项。某程序员自己编写了两个函数库libuser1.a和libuser2.a（库名必须以lib为前缀），当用命令

cc test.c –luser1.a –luser2.a

编译时，报告有未定义的符号，而改用命令

cc test.c –luser2.a –luser1.a

时，能得到可执行程序。试分析原因

（备注：库名中的lib在命令中省略。该命令和命令cc test.c libuser1.a libuser2.a的效果一致）

例　题　5

cc test.c –luser1.a –luser2.a

解答

test.c	libuser1.a	libuser2.a
引用a	定义b	定义a 引用b

例　题　6

cc是UNIX系统上C语言编译命令，–l是连接库函数的选择项。两个程序员分别编写了函数库libuser1.a和libuser2.a，当用命令

cc test.c –luser1.a –luser2.a

编译时，报告有重复定义的符号。而改用命令

cc test.c –luser2.a –luser1.a

时，能得到可执行程序。试分析原因

例　题　6

cc test.c –luser1.a –luser2.a

一种情况

test.c	libuser1.a	libuser2.a
引用a	定义a	定义b
引用b		定义a

若干人一起开发软件时有可能发生

a的使用局部于文件，应加static而未加

例　题　7

两个C文件link1.c和link2.c的内容分别如下

```
int buf[1] ={100};
```

和

```
extern int *buf;
main() { printf("%d\n", *buf); }
```

在X86/Linux经命令cc link1.c link2.c编译后，运行时产生如下的出错信息

Segmentation fault (core dumped)

请说明原因

例　题　7

```
int buf[1] ={100};
```

和

```
extern int *buf;
main() { printf("%d\n", *buf); }
```

- 连接时不检查名字的类型
  - 虽对buf的类型持不同观点，但能连接成目标程序
- 连接时让不同文件中同一名字的地址相同
  - 运行时，在link2.c中，由于buf的内容是100，取*buf的值就是取地址为100的单元的内容。该地址不在程序数据区内，报错
- 若把这些代码放在一个文件中，编译时报错

习　题

第1次：8.11, 8.13, 8.14