


# 共享库和动态链接

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

# 动态链接的共享库（Shared Libraries）

- 静态库有一些缺点：
  - 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
  - 库函数（如printf）被合并在可执行目标中，磁盘上存放着数千个可执行文件，造成磁盘空间的极大浪费
  - 程序员需关注是否有函数库的新版本出现，并须定期下载、重新编译和链接，更新困难、使用不便
- 解决方案: Shared Libraries（共享库）
  - 是一个目标文件，包含有代码和数据
  - 从程序中分离出来，磁盘和内存中都只有一个备份
  - 可以动态地在装入时或运行时被加载并链接
  - Window称其为动态链接库（Dynamic Link Libraries，.dll文件）
  - Linux称其为动态共享对象（ Dynamic Shared Objects, .so文件）

# 共享库（Shared Libraries）

动态链接可以按以下两种方式进行：

- 在第一次加载并运行时进行 (load-time linking).
  - Linux通常由动态链接器(ld-linux.so)自动处理
  - 标准C库 (libc.so) 通常按这种方式动态被链接
- 在已经开始运行后进行(run-time linking).
  - 在Linux中，通过调用 dlopen()等接口来实现
    - 分发软件包、构建高性能Web服务器等

在内存中只有一个备份，被所有进程共享，节省内存空间

一个共享库目标文件被所有程序共享链接，节省磁盘空间

共享库升级时，被自动加载到内存和程序动态链接，使用方便

共享库可分模块、独立、用不同编程语言进行开发，效率高

第三方开发的共享库可作为程序插件，使程序功能易于扩展

# 自定义一个动态共享库文件

myproc1.c

```
# include <stdio.h>
void myfunc1()
{
    printf("%s","This is myfunc1!\n");
}
```

myproc2.c

```
# include <stdio.h>
void myfunc2()
{
    printf("%s","This is myfunc2\n");
}
```

PIC：Position Independent Code

位置无关代码

1）保证共享库代码的位置可以是不确定的

2）即使共享库代码的长度发生变化，也不会影响调用它的程序

```
gcc –c myproc1.c myproc2.c
gcc –shared –fPIC –o mylib.so myproc1.o myproc2.o
```

位置无关的共享代码库文件

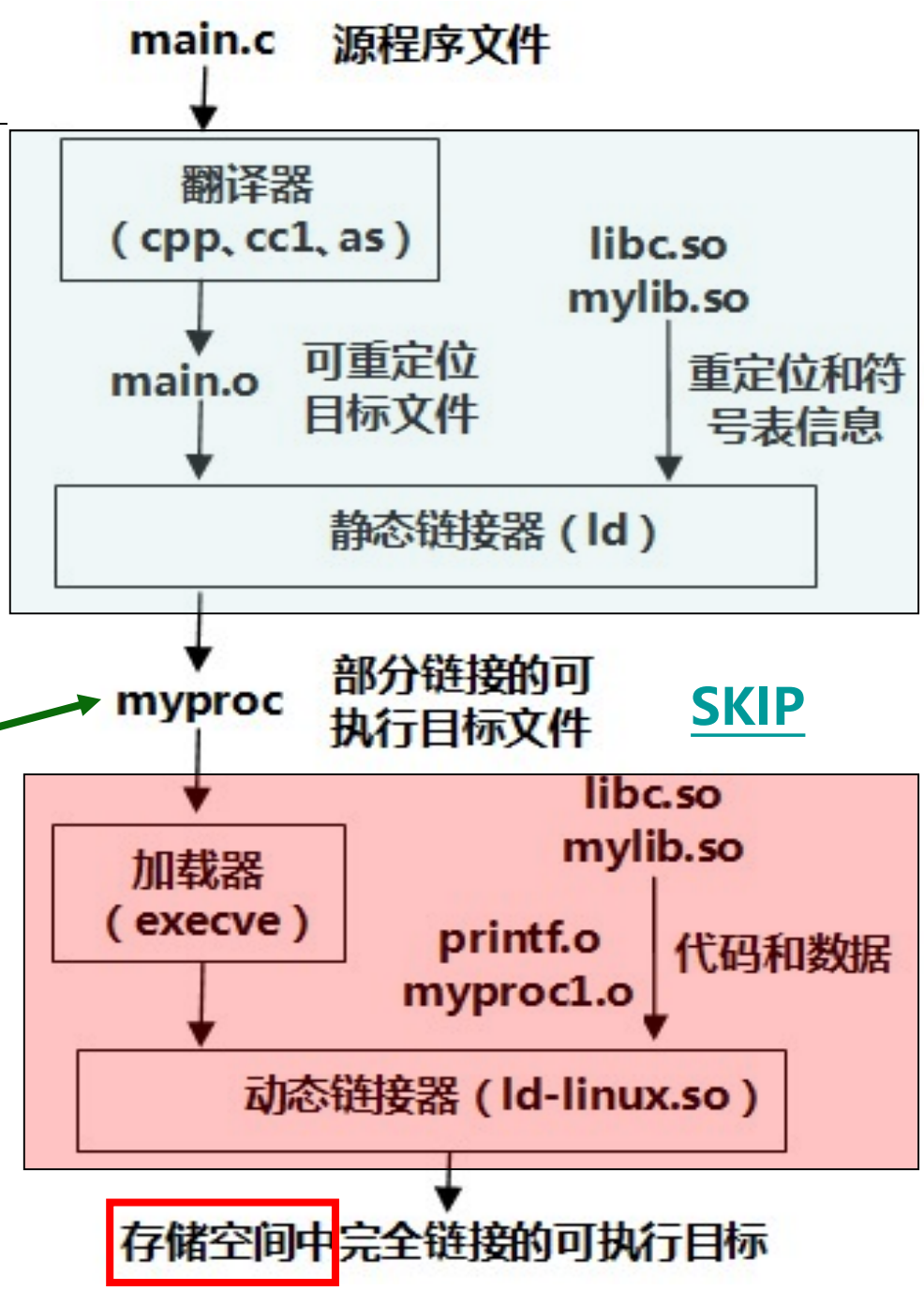
# 加载时动态链接

gcc –c main.c　　libc.so无需明显指出
gcc –o myproc main.o ./mylib.so

调用关系：main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```




加载 myproc 时，加载器发现在其程序头表中有 .interp 段，其中包含了动态链接器路径名 ld-linux.so，因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后，再把控制权交给 myproc，启动其第一条指令执行。

# 加载时动态链接

- 程序头表中有一个特殊的段：INTERP
- 其中记录了动态链接器目录及文件名ld-linux.so

BACK

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

# 运行时动态链

可通过动态链接器接口提供的函数在运行时进行动态链接

类UNIX系统中的动态链接器接口定义了相应的函数，如dlopen, dlsym, dlerror, dlclose等，其头文件为dlfcn.h

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    void *handle;
    void (*myfunc1)();
    char *error;
     /* 动态装入包含函数myfunc1()的共享库文件 */
    handle = dlopen("./mylib.so", RTLD_LAZY);
    if (!handle) {
            fprintf(stderr, "%s\n", dlerror());
            exit(1);
     }
     /* 获得一个指向函数myfunc1()的指针myfunc1*/
     myfunc1 = dlsym(handle, "myfunc1");
     if ((error = dlerror()) != NULL) {
            fprintf(stderr, "%s\n", error);
            exit(1);
     }
     /* 现在可以像调用其他函数一样调用函数myfunc1() */
     myfunc1();
     /* 关闭（卸载）共享库文件 */
     if (dlclose(handle) < 0) {
            fprintf(stderr, "%s\n", dlerror());
            exit(1);
     }
     return 0;
}
```

# 位置无关代码（PIC）

- 动态链接用到一个重要概念：
  - 位置无关代码（Position-Independent Code，PIC
  - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
  - 共享库代码的位置可以是不确定的
  - 即使共享库代码的长度发生变化，也不影响调用它的程序
- 引入PIC的目的
  - 链接器无需修改代码即可将共享库加载到任意地址运行
- 所有引用情况

  (1) 模块内的过程调用、跳转，采用PC相对偏移寻址

  (2) 模块内数据访问，如模块内的全局变量和静态变量

  (3) 模块外的过程调用、跳转

  (4) 模块外的数据访问，如外部变量的访问

要实现动态链接，必须生成PIC代码

要生成PIC代码，主要解决这两个问题

# (1) 模块内部函数调用或跳转

- 调用或跳转源与目的地都在同一个模块，相对位置固定，只要用相对偏移寻址即可
- 无需动态链接器进行重定位

```
8048344 <bar>:
 8048344:   55                  pushl  %ebp
 8048345:   89 e5               movl  %esp, %ebp
 ……
 8048352:   c3                  ret
 8048353:   90                  nop

8048354  <foo>:
 8048354:   55                  pushl %ebp
 ……
 8048364:   e8 db ff ff ff      call 8048344 <bar>
 8048369:
 ……
```

```
static int a;
static int b;
extern  void ext();

void bar()
{
    a=1;
    b=2;
}

void foo()
{
    bar();
    ext();
}
```

call的目标地址为：
0x8048369+
0xfffffdb(-0x25)=
0x8048344

JMP指令也可用相对寻址方式解决

# (2) 模块内部数据引用

- .data节与.text节之间的相对位置确定，任何引用局部符号的指令与该符号之间的距离是一个常数

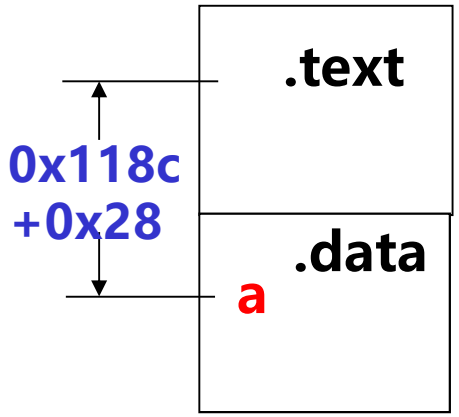
```
static   int a;
extern int b;
extern  void ext();

void bar()
{
    a=1;
    b=2;
}
.......
```

```
0000344 <bar>:
 0000344:   55                      pushl  %ebp
 0000345:   89 e5                   movl   %esp, %ebp
 0000347:   e8 50 00 00 00          call   39c <__get_pc>
 000034c:   81 c1 8c 11 00 00       addl   $0x118c, %ecx
 0000352:   c7 81 28 00 00 00       movl $0x1, 0x28(%ecx)
 ......
 0000362:   c3                      ret

000039c <__get_pc>:
 000039c:    8b 0c 24               movl  (%esp), %ecx
 000039f:    c3                     ret
```

多用了4条指令




变量a与引用a的指令之间的距离为常数，调用__get_pc后，call指令的返回地址被置ECX。若模块被加载到0x9000000，则a的访问地址为：

0x9000000+0x34c+0x118c(指令与.data间距离)+0x28(a在.data节中偏移)

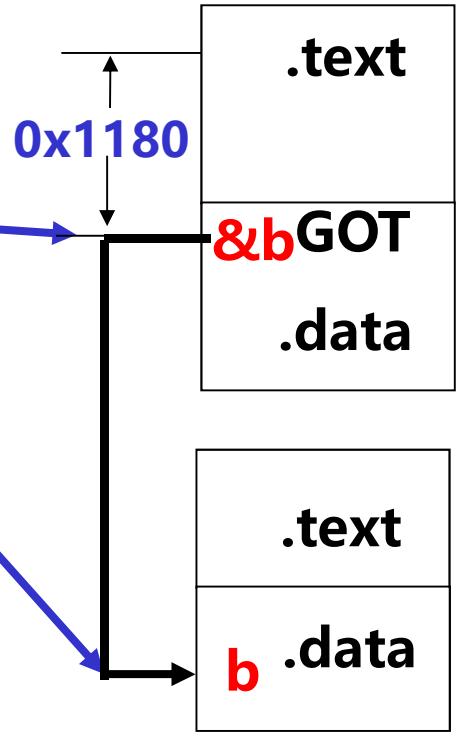
# (3) 模块外数据的引用

- 引用其他模块的全局变量，无法确定相对距离
- 在.data节开始处设置一个指针数组（全局偏移表，GOT），指针可指向一个全局变量
- GOT与引用数据的指令之间相对距离固定

```
00000344 <bar>:
 00000344:    55                      pushl  %ebp
 ......
 00000357:    e8 00 00 00 00          call   0000035c
 0000035c:    5b                      popl   %ebx
 000035d:                             addl   $1180, %ebx
 ......                               movl   (%ebx), %eax
 ......                               movl   $2, (%eax)
 ......
```

```
static   int a;
extern int b;
extern  void ext();

void bar()
{
     a=1;
     b=2;
}
......
```




共享库模块

- 编译器为GOT每一项生成一个重定位项（如.rel节…）
- 加载时，动态链接器对GOT中各项进行重定位，填入所引用的地址（如&b）

PIC有两个缺陷：多用4条指令；多了GOT（Global Offset Table），故需多用一个寄存器（如EBX），易造成寄存器溢出

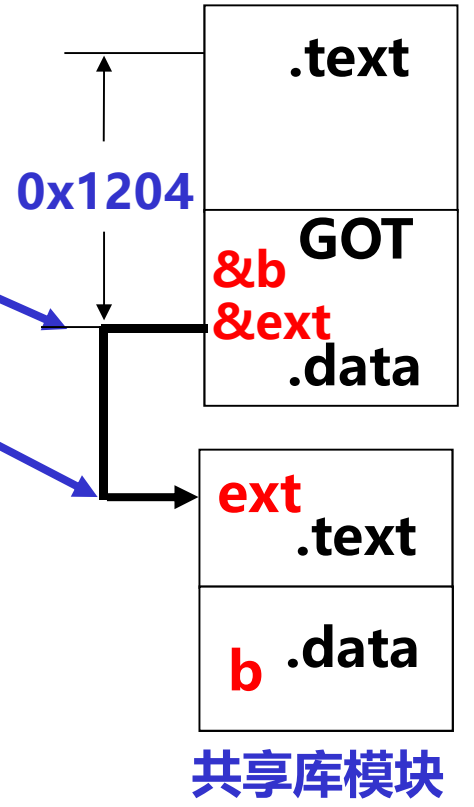
# (4) 模块间调用、跳转

- 方法一：类似于(3)，在GOT中加一个项(指针)，用于指向目标函数的首地址（如&ext）
- 动态加载时，填入目标函数的首地址

```
static   int a;
extern int b;
extern  void ext();

void foo()
{
     bar();
     ext();
}
……
```

```
0000050c <foo>:
 0000050c:    55                      pushl  %ebp
 ……
 00000557:    e8 00 00 00 00          call   0000055c
 0000055c:    5b                      popl   %ebx
 0000055d:                            addl   $1204, %ebx
 ……                                   call   *(%ebx)
 ……
        *(%ebx)为间接地址：R[eip]←M[R[ebx]]
```




- 多用三条指令并额外多用一个寄存器（如EBX）

可用“延迟绑定（lazy binding）”技术减少指令条数：
不在加载时重定位，而延迟到第一次函数调用时，需要用GOT和PLT（Procedure linkage Table, 过程链接表）

# (4) 模块间调用、跳转

```
extern  void ext();
void foo() {
    bar();
    ext();
}
……
```

方法二：延迟绑定

GOT是.data节一部分，开始三项固定，含义如下：

GOT[0]为.dynamic节首址，该节中包含动态链接器所需要的基本信息，如符号表位置、重定位表位置等；

GOT[1]为动态链接器的标识信息

GOT[2]为动态链接器延迟绑定代码的入口地址

调用的共享库函数都有GOT项，如GOT[3]对应ext

延时绑定代码根据GOT[1]和ID确定ext地址填入GOT[3]，并转ext执行，以后调用ext，只要多执行一条jmp指令而不是多3条指令。

PLT是.text节一部分，结构数组，每项16B，除PLT[0]外，其余项各对应一个共享库函数，如PLT[1]对应ext

PLT[0]

```
0804833c:  ff 35 88 95 04 08   pushl  0x8049588
 8048342:  ff 25 8c 95 04 08    jmp   *0x804958c
 8048348:  00 00 00 00
```

PLT[1] <ext>　　用 ID=0 标识ext()函数

```
0804834c:  ff 25 90 95 04 08    jmp  *0x8049590
 8048352:  68 00 00 00 00        pushl  $0x0
 8048357:  e9 e0  ff  ff  ff     jmp    804833c
```

ext()的调用指令：

```
804845b:  e8 ec fe ff ff   call  804834c <ext>
```