Chapter 07 链接

- `C预处理器(cpp)` -> `C编译器(cc1)` -> `汇编器(as)` -> `链接器(ld)`
- **可重定位目标文件**(Relocatable object file) : 在编译时和其它可重定位目标文件合并, 创建可执行目标文件.
- **可执行目标文件**(Executable object file) : 可以被直接复制到内存并执行.
- **共享目标文件**: 可以在 `加载` 或 `运行` 时, 动态地被加载进内存并链接.
- **符号解析**: 把符号引用和符号定义相关联.
- **重定位**: 把符号定义和实际地址相关联.

- **`ELF文件格式`** : Executable and Linkable Format, 可执行可链接格式.
- **`ELF头`** : 描述系统的字大小, 字节顺序, ELF头的大小, 目标文件的类型(可重定位/可执行/共享), 机器类型(x86-64), `节头部表` 的文件偏移, 节头部表中条目的大小和数量.
- **`节头部 表`** (Section header Table): 描述每个节(section)的位置和大小.
- **`节`** (Section):

  `.text` : 已编译程序的机器代码.

  `.rodata` : 只读数据(如switch的跳转表, printf的首参字符串等)

  `.data` : 已初始化的全局变量, 和已初始化的 `static` 变量.

  `.bss` : 未初始化的全局变量和 `static` 变量, 或被初始化为 `0` 的全局变量和 `static` 变量. "bss"可以记忆为"Better Save Space".

  `.symtab` : `符号表` . 存放程序中**定义**和**引用**的 `函数` 和 `全局变量` 的信息.

  `.rel.text` : 一个以 `.text节中的位置` 为元素的列表. (这里的 `rel` 表示 `relocate` , 即 `重定位`).

  `.rel.data` : 被当前模块引用或定义的所有 `全局变量` 的 `重定位` 信息.

  `.strtab` : 一个字符串表, 包括 `.symtab` 中的符号表使用的字符串.

- 对于每个可重定位目标模块 `m` , 都有一个 `符号表` (`.symtab`). 它包含 `m` **定义**和**引用**的符号的信息.
  - **全局符号**: 由m定义, 且可以被其他模块引用的符号. 对应于 `非static的` 函数和全局变量.
  - **外部符号**: 由其他模块定义, 并被m引用的符号. 对应于其他模块定义的 `非static的` 函数和全局变量.
  - **局部符号**: 由m定义, 能且仅能在m中任何位置可见. 对应于m定义的 `static的` 函数和全局变量.
- **`ELF符号表`** 是一个如下格式的条目的数组.

```
struct Elf64_Symbol{
    int name;                // 在字符串表中的字节偏移
  char type:4,      // 是数据? 还是函数?
         binding:4; // 本地的? 还是全局的?
  char reserved;    // *Unused*
  short section;    // 存储在哪个节(section)? (本质是节头部表数组的索引,伪节
```

```
    UNDEF,COMMON,ABS除外）
7     long value;           // 距离这个节的起始地址偏移量是多少？（COMMON: 对齐要求）
8     long size;            // 目标的大小？（COMMON: 目标的最小大小）
9   };
```

- **伪节(pseudo_section)**:
  - ABS: 不该被重定位的符号.
  - COMMON: 见下文.
  - UNDEF: 在本模块中被引用, 但是是在其他地方定义的符号.
- `COMMON` 和 `.bss` 的区别:
  - `COMMON` : 未初始化的全局变量. (**弱全局符号**)
  - `.bss` : 初始化为0的全局变量; 以及未初始化或初始化为0的static变量. (即: **强全局符号**; 以及只有m本身可见故不涉及强弱的**static**)

- 对 `多重定义的全局符号` 的处理方式:
  - 全局符号的强弱(Strong/Weak):
    - 强全局符号: 函数和已初始化的全局变量
    - 弱全局符号: 未初始化的全局变量
  - 规则:
    - 不允许多个同名的强符号
    - 如果有一个强符号和多个弱符号同名, 选择强符号
    - 如果有多个弱符号同名, 任取一个(这里的任意性可能造成越界bug, 比如 `double a;` 和 `int a;`)
- 这里就可以知道为什么要区分 `COMMON` 和 `.bss` 了:
  - 当编译器翻译某个模块m时, 如果遇到一个弱全局符号 `x` , 它是不能知道是否有其他模块定义了 `x` , 所以编译器会把 `x` 分配成 `COMMON` , 把选择权留给链接器.
- 当编译器遇到一个初始化为 `0` 的全局符号y, 由于指定了初始值, 这是一个强符号. 所以编译器可以确信 `y` 要分配到 `.bss` 中去.
- **READELF 程序**: 查看目标文件内容的很方便的工具.

- **静态库(Static Libarary)**: 后缀为 `.a` , 代表"存档(archive)".
- 在符号解析阶段, 链接器 `从左到右` 按照命令行上的出现顺序扫描 `可重定位文件(.o)` 和 `存档文件(.a)` .
  具体地, 链接器维护三个集合:
  - 集合E(Execute): 这个集合的文件用于合并生成可执行文件.
  - 集合U(Undefined): 被文件引用了, 但暂时还未被解析(定义)的符号集.
  - 集合D(Defined): 在前面的输入文件中已经定义了的符号的集合.

对于命令行上的文件f, 如果f是 `可重定位文件(.o)` , 那么所有f中定义的符号都会被加入到E中, 并修改相应的U和D; 而如果f是 `静态库存档文件(.a)` , 那么只有那些在当前U中存在的符号才会被加入到E中, 并修改相应的U和D, 其余的符号直接被舍弃.

如果对命令行上的文件扫描完毕后, U是空集, 此时链接器会合并和重定位E中的文件, 生成可执行文件;

而如果U非空, 说明有未被定义的符号, 说明出现了错误, 链接过程输出错误信息并终止.

- 由于库的这种性质(只有在它左边的文件需要的符号被加入, 其余被直接丢弃), 一般而言库会放在命令行的末尾. 如果库不是相互独立的, 那么就需要适当地排列这些库, 来解决他们之间的依赖关系. (库可以在命令行中重复出现)

- **重定位条目**: 当 `汇编器` 遇到对最终内存位置未知的内存引用, 就会生成一个重定位条目, 放在 `.rel.text` 中(对于代码)和 `.rel.data` 中(对于已初始化的数据) 所以说 `链接器` 在这里只是无脑地按照 `汇编器` 的指示来工作.
- `重定位条目` 的格式:

```
struct Elf64_Rela{
    long offset;        // 被修改的引用, 相对于 **其所在节(section)** 的偏移量
  long type:32,     // 重定位类型, 最基本的是R_X86_64_PC32和R_X86_64_32两种, 即PC相对地址和绝对地址.
         symbol:32; // 被修改的引用指向的符号.
  long addend;      // 有符号常数, 对被修改引用的值做偏移调整.
};
```

- 重定位**PC相对引用**:

```
r.offset = 0xf;
r.symbol = sum;
r.type   = R_X86_64_PC32;
r.addend = -4;  // 比如指令是 e8 00 00 00 00 , 那修改位置是第一个00, 而下一条指令的位置在4字节之后.
```

上例表示: 修改偏移量 `0xf` 处的32位PC相对引用, 使它在运行时会指向 `sum` 函数.

- 重定位**绝对引用**:

```
r.offset = 0xa;
r.symbol = array;
r.type   = R_X86_64_32;
r.addend = 0;
```

上例表示: 修改偏移量 `0xa` 处的32位绝对引用, 使它在运行时会指向 `array` 数组(的第一个字节).

- **可执行目标文件**
- **加载**可执行目标文件:
  - **加载器**: 任何Linux程序都可以通过调用 `execve` 函数来调用加载器.

加载器将可执行目标文件中的代码和数据从磁盘复制到内存中, 然后跳转到第一条指令(入口点)来运行该程序.

- Linux x86-64中, `代码段` 总是从地址 `0x400000` 处开始, 后面是 `数据段`;
  再后面是 `运行时堆`, 它通过调用 malloc 库向上增长.
  堆后面的区域是为 `共享模块` 保留的.
  `用户栈` 从最大的合法用户地址(即 $2^{48} - 1$ 以下) 开始, 向下生长.
  而栈之上的区域(即 2^{48} 开始以上) 是为内核(Kernel)中的代码和数据保留的.

内核内存
用户栈 ⏬
共享库的内存映射区域
⏫ 运行时堆(由malloc创建)
读/写段(.data, .bss)
只读代码段(.init, .text, .rodata)

- **(执行前, 加载时)动态链接共享库**:
  也叫做 `共享目标`, 在Linux中以 `.so` 为后缀. (在Windows中也有类似的东西, 称作 `DLL(动态链接库)`)
  当创建可执行文件时, 执行静态链接, 当程序加载(磁盘->内存)时, 才动态完成链接过程:
  即没有任何 `.so` 的代码或数据真正被复制到了可执行文件中;
  反之, 链接器复制了一些重定位和符号表信息, 使得运行时可以解析对 `.so` 中的代码和数据的引用.

- **(运行时)从应用程序中加载和链接共享库**:
  Linux系统为动态链接器提供了接口, 允许应用程序在运行时加载和链接共享库:

```
1 #include<dlfcn.h>
2 void* dlopen(const char* filename, int flag); // 若成功则返回指向句柄的指针, 否则返回
  NULL
3 void* dlsym(void* handle, char* symbol);            // 若成功则返回指向符号的指针, 否则
  返回NULL
4 void* dlclose(void* handle);    // 若成功则返回0, 否则返回-1
5 const char* dlerror();      // 若之前对上述三个函数的调用产生失败, 则返回错误信息. 若均成功,
  返回NULL.
```

- **位置无关代码(Position Independent Code, PIC)**:

  可以加载而无需重定位的代码. (共享库的编译必须使用该选项! (`-fpic`))

- PIC**数据引用**:

  原理: 无论在内存的何处加载某个模块, 其 `数据段` 和 `代码段` 的**相对距离**总是保持不变的.

  具体的做法: 创建**全局偏移量表(Global Offset Table, GOT)**. 当需要引用全局变量时, 代码中使用PC相对寻址(即在第三章令cpy迷惑的%rip相对寻址!), 通过GOT进行间接引用.

  而在加载时, 动态链接器会重定位GOT中的每个条目, 使得它存储正确的绝对地址.

- PIC**函数调用**:

  延迟绑定(Lazy Binding): 将过程地址的绑定推迟到第一次调用该过程时. 这是因为一个共享库中可能有成百上千个函数, 但我们一般只会用其中很少的一部分. 避免成百上千个并不需要的重定位.

  这样虽然第一次调用过程的 `运行时开销` 很大, 不过其后的每次调用只会花费一条指令和一次内存间接引用的时间.

  过程链接表(Procedure Linkage Table, PLT): 是一个条目数组, 其中每个条目是一个16字节的代码. PLT[0]是一个特殊条目, 用于跳转到动态链接器中, 实现对GOT数组的特定位置的重写, 这样下次调用时就是直接调用相应的函数, 而不是通过PLT[0]调用动态链接器.

- **库打桩机制**: (没看懂, 因为cpy没仔细看)

  这一机制允许截获对共享库函数的调用, 取而代之执行自己的代码.

  - 编译时打桩
  - 链接时打桩
  - 运行时打桩