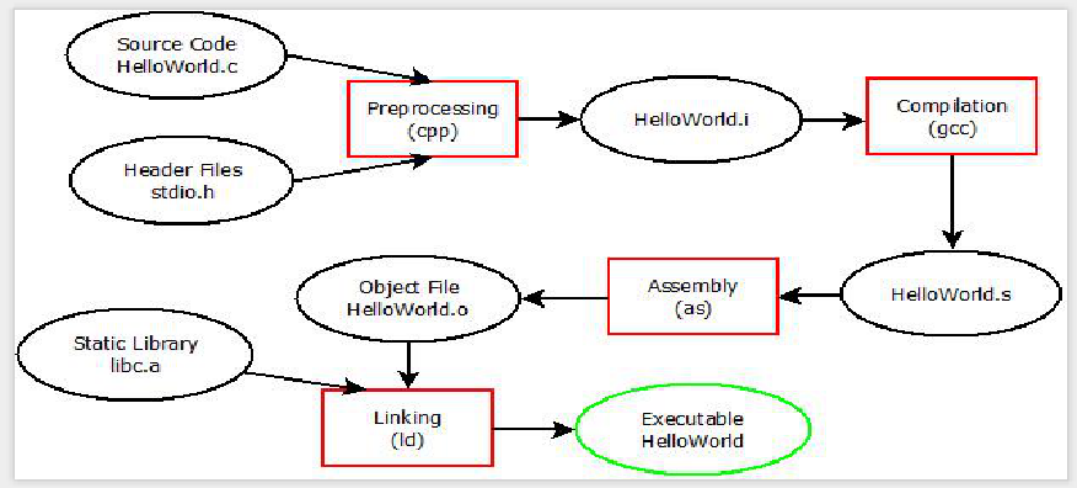
Linux环境下：程序的链接，装载和库

课程来源：BV1hv411s7ew

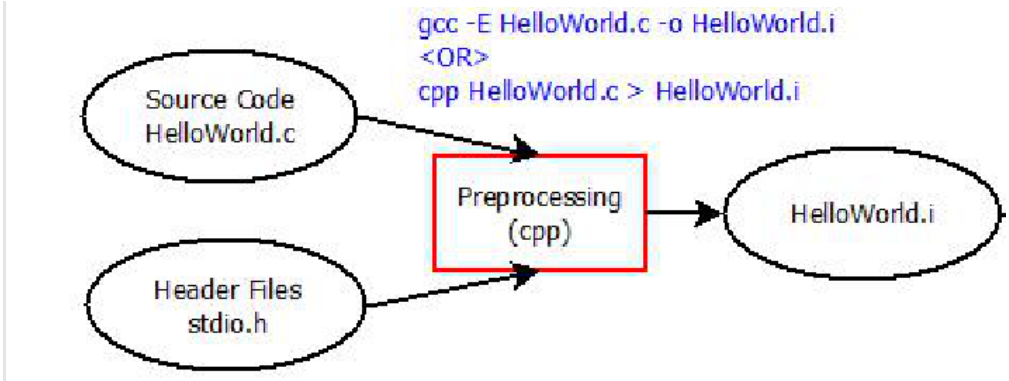
参考：BV1fG411K7Yv

第1部分 编译过程总体概览

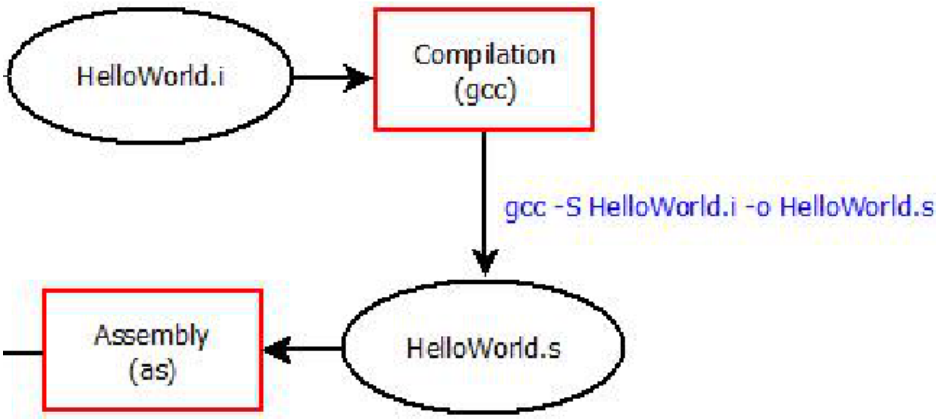
- GCC编译过程
  - 预处理/预编译
  - 编译
  - 汇编
  - 链接



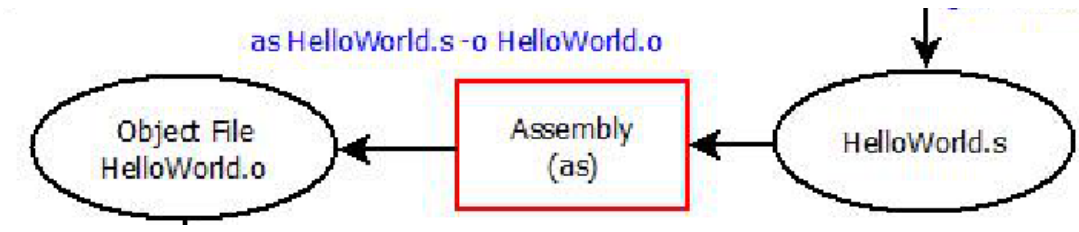
- 预编译
  - 步骤



        - 处理`#define`，将所有的宏定义`#define`展开
        - 处理`#if#else` `#endif`等条件编译指令
        - 处理`#include`原地插入文件
        - 删除注释
    - 调用方法
        - `cpp HelloWorld.c > HelloWorld.i` 或 `gcc -E HelloWorld.c -o HelloWorld.i`
    - 作用
        - 可通过查看预编译结果确认头文件的包含或者宏定义的问题
- 编译
    - 过程

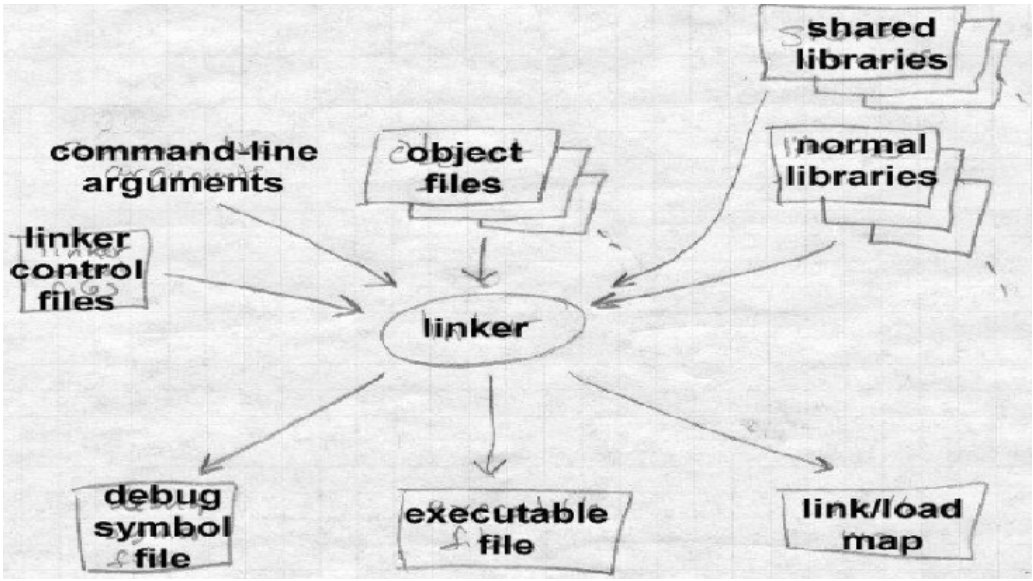


        - 对预编译完的文件进行词法，语法，语义分析并进行优化，最终产生对应的汇编代码文件
    - 调用方法
        - `gcc -S HelloWorld.i -o HelloWorld.s`
- 汇编
    - 过程



        - 把汇编代码转换成目标文件
    - 调用方法

        - `as HelloWorld.s -o HelloWorld.o`
        - 注意
            - 生成的`.o`文件是ELF格式，直接用vim打开会乱码（详见第2部分）
            - 应使用`objdump`查看`objdump -h HelloWorld.o`
- 链接
    - 过程



        - 链接器将多个目标文件链接起来形成可执行文件
        - 比如`HelloWorld.c`中的`main`函数并不知道`printf`这个函数的地址，链接器在链接的时候会根据引用到的符号`printf`，自动去相应的模块查找`printf`的地址，然后将`HelloWorld.c`模块中引用到`printf`的指令进行重新修正，让它的目标地址成为真正的`printf`函数的地址
    - 分类
        - 静态链接
            - 生成的可执行文件大小更大
        - 动态链接

第2部分 目标文件

- ELF格式(Executable Linkage Format)
    - 可执行文件

    - 目标文件
    - 动态链接库 `.so` 和核心转储文件 `core dump`
- 怎么查看目标文件
    - `objdump -h xxxx.o` 可查看各个段的分布

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000041  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000084  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  0000008c  2**2
                  ALLOC
  3 .rodata       00000003  0000000000000000  0000000000000000  0000008c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000012  0000000000000000  0000000000000000  0000008f  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000a1  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000a8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

    - `objdump -s -d xxxx.o` 可挖掘各个段的内容
- 目标文件是什么样
    - 代码段 `.text`
    - 数据段 `.data`
        - 保存初始化的全局变量和局部静态变量
    - 只读数据段 `.rodata`
        - 保存只读数据，一般是只读变量和字符串常量
    - BSS段 `.bss`
        - 保存未初始化的全局变量和局部静态变量
    - Comment段 `.comment`
        - 存放编译器的版本信息
- ELF头
    - `readelf -h xxxx` 查看ELF文件头

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          360 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

- ELF段表
  - `readelf -S xxxx` 查看ELF段表
    - 之前使用 `objdump -h` 命令的时候其实只是显示了关键的段而省略了一些辅助性的段

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000000000  00000040
       0000000000000041  0000000000000000  AX       0     0     4
  [ 2] .rela.text        RELA            0000000000000000  00000678
       0000000000000048  0000000000000018          11     1     8
  [ 3] .data             PROGBITS        0000000000000000  00000084
       0000000000000008  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS          0000000000000000  0000008c
       0000000000000004  0000000000000000  WA       0     0     4
  [ 5] .rodata           PROGBITS        0000000000000000  0000008c
       0000000000000003  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS        0000000000000000  0000008f
       0000000000000012  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000  000000a1
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS        0000000000000000  000000a8
       0000000000000058  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA            0000000000000000  000006c0
       0000000000000030  0000000000000018          11     8     8
  [10] .shstrtab         STRTAB          0000000000000000  00000100
       0000000000000061  0000000000000000           0     0     1
  [11] .symtab           SYMTAB          0000000000000000  000004a8
       0000000000000180  0000000000000018          12    11     8
  [12] .strtab           STRTAB          0000000000000000  00000628
       000000000000004c  0000000000000000           0     0     1
```

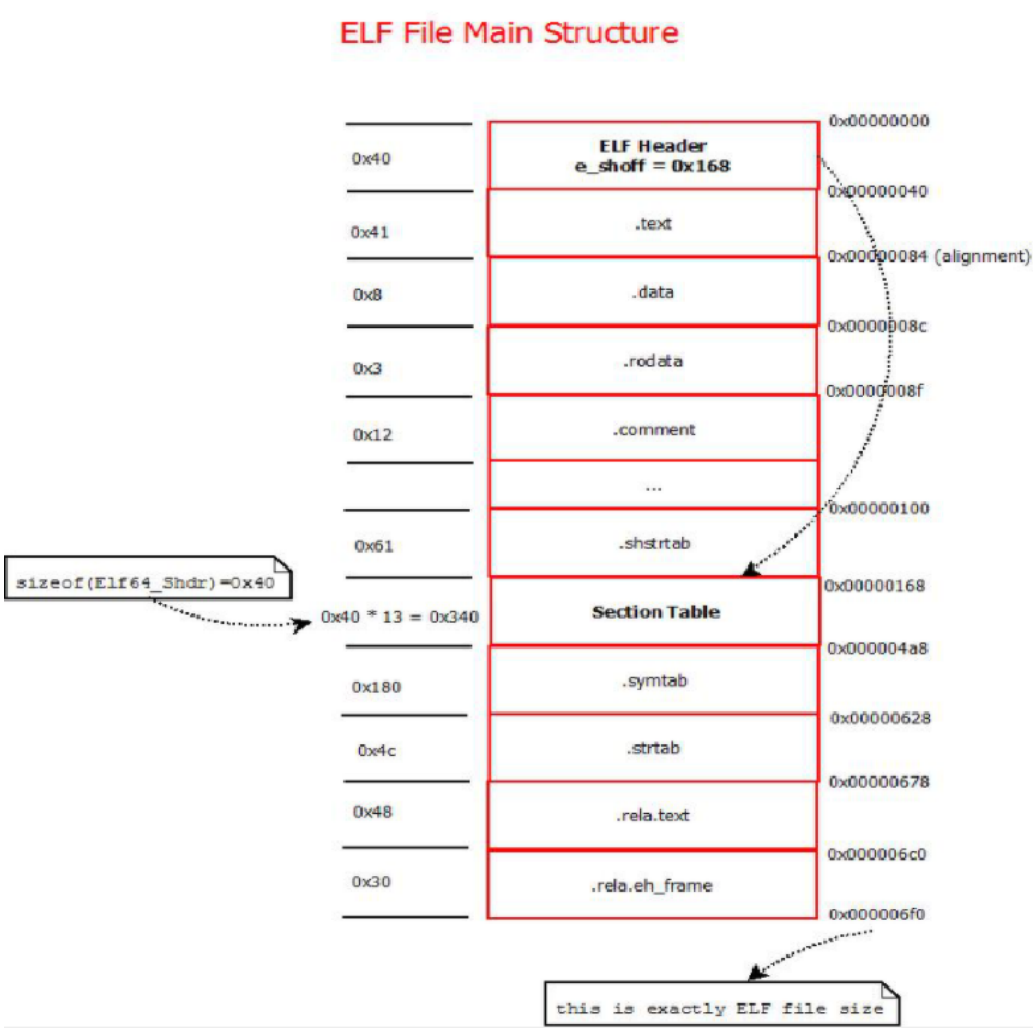
- 字符串表
  - `readelf -p section_name xxxx` 查看字符串表
  - ELF文件里面都用字符串表的偏移来索引字符串

```
String dump of section '.strtab':
  [     1]  c_code_obj.c
  [     e]  var4.2184
  [    18]  var3.2183
  [    22]  g_init_var1
  [    2e]  g_uninit_var2
  [    3c]  foo
  [    40]  printf
  [    47]  main
```

- 符号表
  - `readelf - s xxxx` 查看符号表

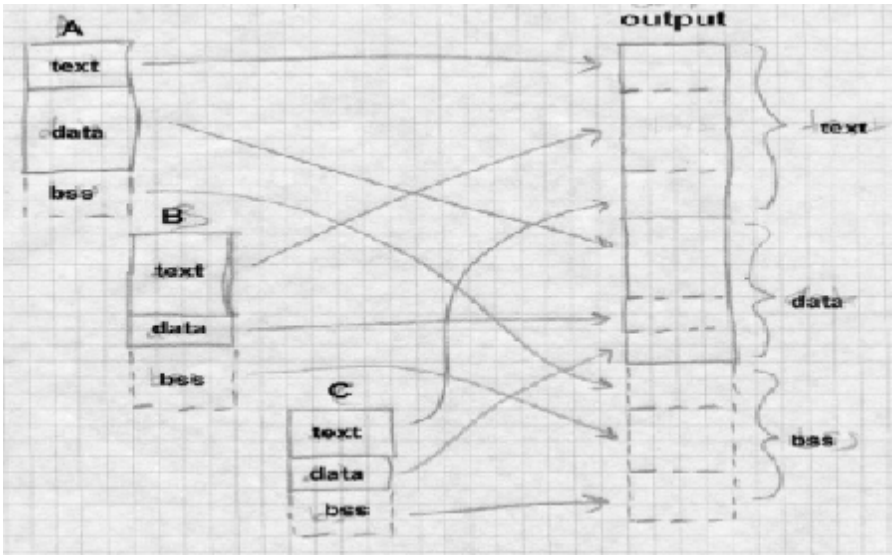
```
Symbol table '.symtab' contains 16 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS c_code_obj.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 var4.2184
     7: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 var3.2183
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    11: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 g_init_var1
    12: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM g_uninit_var2
    13: 0000000000000000    33 FUNC    GLOBAL DEFAULT    1 foo
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    15: 0000000000000021    32 FUNC    GLOBAL DEFAULT    1 main
```

- ELF文件总体结构图



第3部分 静态链接

- 空间和地址分配
  - 链接器 `ld`
    - `ld main.o swap.o -e main -o stlink`



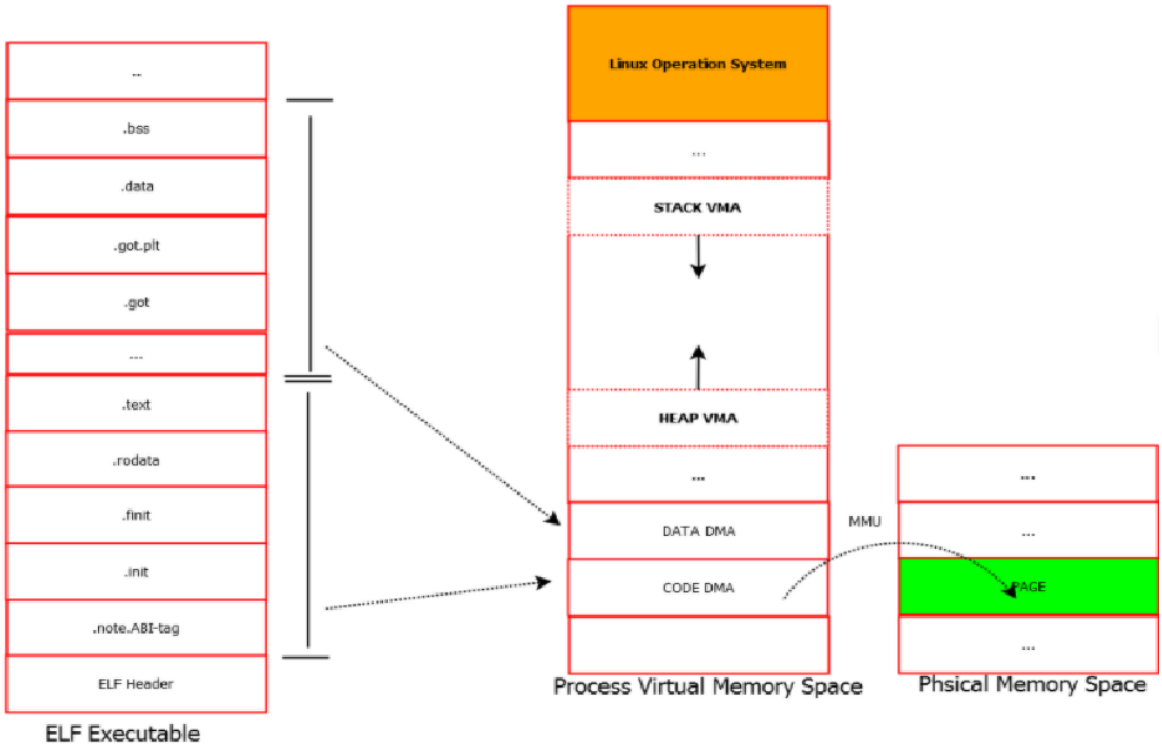
- 链接时重定位
  - 参数：绝对寻址
  - 函数：相对寻址
  - 链接器怎么知道那些符号要重定位呢?
    - 重定位表
    - `objdump –r main.o`

第4部分 可执行文件的装载

- 程序(可执行文件)和进程的区别
  - 静态 vs 动态
- 现代操作系统如何装载可执行文件
  - 给进程分配独立的**虚拟地址空间**
  - 建立虚拟地址空间和可执行文件的映射关系
  - 把CPU指令寄存器设置成可执行文件的入口地址，启动执行
- 可执行文件在装载的过程中是映射的虚拟地址空间
  - 所以可执行文件通常被叫做映像文件（或者Image文件）
- ELF文件的两种视角
  - 区段（section）
    - 从链接器、编译器、汇编器的视角来看

    - 段（segment）
        - 从系统加载器执行的视角来看，也就是它会被映射到内存中。
    - 一个段（segment）由多个区段（section）组成
        - 例如，一个“可加载只读”段由可执行代码区段、只读数据区段和动态链接器需要的符号组成

- ELF可执行文件在Linux下的装载过程



- 操作系统在进程启动前将系统环境变量和进程的运行参数提前保存到进程栈当中，进程启动之后，把这些参数信息传递给main函数，即我们熟知的argc和argv两个参数
- 深入Linux内核探寻它是如何识别和装载ELF文件
    - 在bash下输入命令：执行某一个ELF文件
    - bash进程调用 `fork()` 系统调用创建一个新的进程
    - 新的进程调用 `execve()` 系统调用执行指定的ELF文件
    - 检查ELF文件头部信息
    - 加载程序头表
    - 寻找和处理解释器段
    - 装入目标程序的段
    - 填写目标程序的入口地址

    - 填写目标程序的参数，环境变量等信息
    - 将eip和esp改成新的地址，使得CPU在返回用户空间时就进入新的程序入口

第5部分 动态链接

- 程序在链接时优先链接动态库
    - 除非显式地使用 `-static` 参数指定链接静态库
    - `gcc -o sta_main -static main.c`
- 静态和动态链接的可执行文件大小差距显著
    - 静态库被链接后库直接嵌入可执行文件中
        - PROS：
            - 可移植性强，所有的功能都封到二进制文件中了，不需要依赖系统是否有合适的lib库
        - CONS：
            - 浪费系统空间
            - 改动库需要找出链接该库的所有程序，并重新编译。
    - 动态库是在程序运行时被链接的
        - CONS：
            - 如果把二进制文件放到没有安装对应的 `libc` 的linux中会报共享库缺失的错误
            - 共享库需要有很好的兼容性。
- 动态链接对象
    - 共享对象(Shared Object)
    - `.so`
- 假设有两个程序 `main_1.c` 和 `main_2.c` 需要调用动态链接库 `hello.so` 中的同一个函数
    - 生成动态链接库 `hello.so`
        - `gcc -fPIC -shared -o hello.so hello.c`
        - `-shared` 表示产生共享对象，`-fPIC` 表示产生位置无关代码
    - 在编译时链接动态链接库
        - `gcc -o main_1 main_1.c ./hello.so`

        - `gcc -o main_2 main_2.c ./hello.so`
- 装载时重定位
    - 编译完成后最终的装载地址不确定
    - 技术
        - PIC位置无关代码
        - 延迟绑定
- 引入动态链接后，在操作系统开始运行应用程序之前，首先会把控制权交给动态链接器，它完成了动态链接的工作之后再把控制权交给应用程序
- 使用自己写的共享库
    - 符合命名规范 `libhello.so`
        - `mv hello.so libhello.so`
        - `sudo mv libhello.so /usr/lib/`
    - 使用
        - `gcc -o main_1 main_1.c -lhello`
    - 如果生成的可执行文件提示找不到该so文件，运行 `ldconfig`
        - 这个程序的作用是为共享库更新符号链接
            - 通过 `ldd` 指令可以看到动态链接的库名，和文件路径

```
[root@LocalCentOS dynamic_example]# ./main
./main: error while loading shared libraries: libhello.so: cannot open shared object file: No such file or directory
[root@LocalCentOS dynamic_example]# ldd main
        linux-vdso.so.1 =>  (0x00007fffd45b7000)
        libhello.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x0000003fcd600000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003fcd200000)
```

```
[root@LocalCentOS dynamic_example]# ldconfig
[root@LocalCentOS dynamic_example]# ldd main
        linux-vdso.so.1 =>  (0x00007fff8a9ff000)
        libhello.so => /usr/lib/libhello.so (0x00007f5c1bbe7000)
        libc.so.6 => /lib64/libc.so.6 (0x0000003fcd600000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003fcd200000)
```

第6部分 入口函数和运行库

- 程序并非从 `main` 函数开始执行
    - 操作系统装载程序之后首先运行的代码是某些运行库的代码，它们负责初始化 `main` 函数正

常执行所需要的环境，负责调用 `main` 函数，并记录 `main` 函数的返回值，调用 `atexit` 注册的函数，最后结束进程

- 运行库和平台相关，与操作系统联系紧密
  - 运行库可理解成C/C++程序和操作系统之间的抽象层，使得大部分时候程序不用直接和操作系统的API和系统调用打交道
  - 运行库把不同的操作系统API抽象成相同的库函数，方便应用程序的使用和移植