链接与加载选讲

蒋炎岩

南京大学　　计算机科学与技术系　　计算机软件研究所

本讲概述

动态链接，大家明白了吗?
(根据我们的经验，大家上课没听懂)

本讲内容

- 静态链接与加载
  - Hello 程序的链接与加载
- 动态链接与加载
  - 自己动手实现动态加载

静态链接与加载

一个实验：-fno-pic 编译；-static 链接

代码

```c
// a.c
int foo(int a, int b) {
  return a + b;
}
```

```c
// b.c
int x = 100, y = 200;
```

```c
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
  printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

GNU Binutils

Binary utilities: 分析二进制文件的工具

- RTFM: 原来有那么多工具！
  - 有 addr2line，自己也可以实现类 gdb 调试器了

使用 binutils

- objdump 查看 .data 节的 x, y (`-D`)
- objdump 查看 main 对应的汇编代码
- readelf 查看 relocation 信息
  - 思考题：为什么要 `-4`?

静态程序的加载

由操作系统加载(下学期内容)

你可以认为是“一步到位”的

- 使用 gdb (starti) 调试
- 使用 strace 查看系统调用序列

一个有趣的问题："最小"的可执行代码

不能用 ld 链接吗?

(试一试)

- 我们能否写一个最小的汇编代码，能正确返回？
- 仅执行一个操作系统调用
  - rax = 231; rdi = 返回值
  - syscall 指令执行系统调用

Linux/C 世界中的宝藏

`gcc -Wl,--verbose`

C 世界里还有很多大家不知道的东西

- 一系列 crt 对象
  - `__attribute__((constructor))`
  - `__attribute__((destructor))` (atexit)

RTFM; RTFSC!

动态链接与加载

去掉 `-fno-pic` 和 `-static`

这是默认的 gcc 编译/链接选项。

文件相比静态链接大幅瘦身

- `a.o`, `b.o` 没有变化
- `main.o` 里依然有 `00 00 00 00`
  - 但是相对于 rip 的 offset
  - relocation 依然是 x - 4, y - 4, foo - 4
- `a.out` 里库的代码都不见了

位置无关代码

使用位置无关代码 (PIC) 的原因

- 共享库不必事先决定加载的位置
- 应用程序自己也是
  - 新版本 gcc 无论 32/64-bit 均默认 PIC
  - 重现课本行为可使用 `-no-pie` 选项编译

PIC 的实现

- i386 并不支持以下代码

```
movl $1, 1234(%eip)
```

- 于是有了你们经常看到的 `__i686.get_pc_thunk.bx`
  - “获取 next PC 的地址” (如何实现？)

共享对象 (a.out) 的加载

命令：ldd

- “print shared object dependencies”
  - linux-vdso.so.1
    - 暂时忽略它
  - libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
  - /lib64/ld-linux-x86-64.so.2
    - 多次打印，地址会发生变化 (ldd 会执行加载过程)

ldd 竟然是一个脚本 (`vim $(which ldd)`)

- 挨个尝试调用若干 ld-linux.so 候选
  - 加上一系列环境变量
  - 我们可以用 `--list` 选项达到类似效果

共享对象 (a.out) 的加载 (cont'd)

终于揭开谜题

- `readelf -a a.out`
  - program header 中有一个 INTERP
    - `/lib64/ld-linux-x86-64.so.2`
    - 这个字符串可以直接在二进制文件中看到

神奇的 ld.so

- `/lib64/ld-linux-x86-64.so.2 ./a.out`
  - 和执行 `./a.out` 的行为完全一致
    - (这才是 `./a.out` 真正在操作系统里发生的事情)
      - 与 sha-bang (`#!`) 实现机制完全相同
- ld.so 到底做了什么，下学期分解

动态链接：实现

我们刚才忽略了一个巨大的问题

main.o 在链接时，对 printf 的调用地址就确定了
但 libc 每次加载的位置都不一样啊!

应用程序使用怎样的指令序列调用库函数？

- 可以在库加载的时候重新进行一次静态链接
  - 但这个方案有一些明显的缺陷
    - 各个进程的代码不能共享一个内存副本
    - 没有使用过的符号也需要重定位

ELF: 查表

基本款

```
call *table[PRINTF]
```

- 在链接时，填入运行时的 table
  - 使用 `-fno-plt` 选项可以开启此款

豪华款 (默认选项，使用 PLT)

```
printf@plt:
  jmp *table[PRINTF]
  push $PRINTF
  call resolve
```

总结与反思

为什么我从来没听说过这些知识？？？

> 因为百度搜不到啊!

中文社区几乎不存在这些知识的详细解释

- the friendly manual 是英文写的
- mailing list/stackoverflow 都是英文
- 国内的专家本来就少
  - 仅有的那些也没空写文档

你们才是未来的希望

- 不要动不动就说内卷
- 不要为了无意义的 GPA 沾沾自喜
- 不要停止自我救赎

End.

(RTFM; STFW; RTFSC)