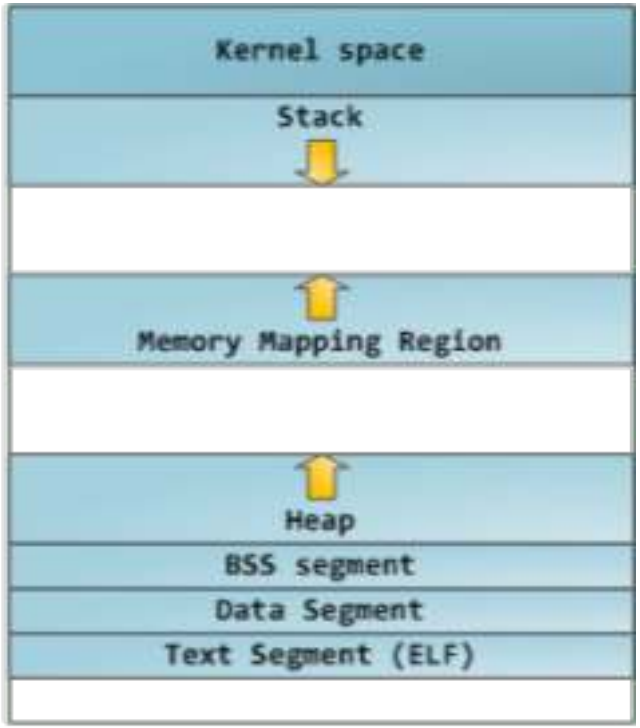
链接与加载

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

该回答“程序是如何在OS上执行的”了

- 需要的成员都到齐了
  - 进程抽象：执行的程序、系统调用
  - 虚存抽象：地址空间中映射的区域
  - 应用程序：busybox
  - 库函数：newlib



回到那个经典的问题

- 在Shell里运行printf(“Hello World\n”)程序，从软件到硬件都发生了什么？

- 运行从Shell开始
  - 用fork()创建一个新进程
  - 用execve()执行Hello World的二进制文件
  - 子进程开始执行

- 从execve之后的第一条指令开始，软件和硬件都发生了什么？

简单的情况

简单的情况

- gcc –static hello.c
- 编译成静态二进制文件
  - ldd报错： “not a dynamic executable”

execve执行的到底是什么？

- 程序执行的第一条指令来自哪里?
  - main
    - C语言规定的入口
  - section .init中的_init
    - objdump能看到
  - 代码节的_start
    - readelf –h的entry
- 如果我想知道execve之后是从哪里执行的，怎么办?

简单的情况

- 用gdb观察指令执行
  - (想办法在执行第一条指令之前停下来)
  - 第一条指令：0x0000000000400892 in _start ()
  - backtrace (bt)：只有这一条记录

- So far, so good
  - 从ELF二进制文件中的entry开始执行(readelf –h可以得到)
  - 调用libc中的代码
    - libc代码调用main(int argc, char **argv, char **envp)
    - 在main返回后完成收尾工作(执行atexit注册的函数)
    - 执行_exit(retval)

printf("Hello, World\n");

- 用objdump查看编译出的指令
  - 完全没有调用printf
    - `4009b2: bf 84 10 4a 00  mov  $0x4a1084,%edi`
    - `4009b7: e8 d4 f0 00 00  callq  40fa90 <_IO_puts>`
    - `4a1084: rodata “Hello, World\n”`
  - puts (_IO_puts)属于C标准库
    - 可以参考newlib中的代码
    - 经历了一系列操作，调用write打印了Hello World

strace的结果

- Hello World到底在操作系统层做了什么？

```
execve("./a.out", ["./a.out"], ...)     = 0
uname({sysname="Linux", ...})           = 0
brk(NULL)                               = 0x1c1c000
brk(0x1c1d1c0)                          = 0x1c1d1c0
arch_prctl(ARCH_SET_FS, 0x1c1c880)      = 0
readlink("/proc/self/exe", ..., 4096)   = 51
brk(0x1c3e1c0)                          = 0x1c3e1c0
brk(0x1c3f000)                          = 0x1c3f000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
fstat(1, {st_mode=S_IFCHR|0620, ...})   = 0
write(1, "Hello World\n", 12)           = 12
exit_group(0)                           = ?
```

真的是简单的情况?

- 如果是a.cpp
- 何时打印Hello World?

```
1 #include <cstdio>
2
3 class A {
4 public:
5  A() {
6   printf("Hello, World\n");
7  }
8 };
9
10 A a;
11
12 int main() {
13  return 0;
14 }
```

为什么用C语言写操作系统实验?

- 你也许不完全理解C++在-ffreestanding -no-rtti的行为
  - 虚函数?
  - 全局变量的构造函数?
  - 异常? dynamic_cast?

```
#0  0x080488d1 in A::A (this=0x80eaf9c <a>) at a.cpp:6
#1  0x080488a8 in __static_initialization_and_destruction_0
(__initialize_p=1,     __priority=65535) at a.cpp:10
#2  0x080488c3 in _GLOBAL__sub_I_a () at a.cpp:14
#3  0x080493ad in __libc_csu_init ()
#4  0x08048abe in generic_start_main ()
#5  0x08048d0d in __libc_start_main ()
#6  0x08048757 in _start ()
```

实际的情况

动态链接库

- gcc –o hello hello.c
  - hello二进制文件动态链接libc
- 观察strace的结果
  - Hello World程序执行的系统调用多了很多
- 万事总要开头
  - 还是从执行的第一条指令开始

GDB告诉我们答案

- 调用栈
  - 0x00002aaaaaaabc33 in _start () from /lib64/ld-linux-x86-64.so.2
  - 0x0000000000000001 in ?? ()
  - 0x00007fffffffe7db in ?? ()
  - 0x0000000000000000 in ?? ()

- 这是操作系统在执行execve时帮我们完成的吗?

Hello, OS World

- 执行execve以后，查看了一下可用内存

```
execve("./a.out", ["./a.out"], [/* 27 vars */]) = 0
brk(NULL)                                       = 0x1a0d000
```

- 试图访问ld.so.nohwcap

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
```

- 没这个文件；申请8KB的空间，可读可写

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4db691000
```

Hello, OS World

- ld.so.preload?

```
access("/etc/ld.so.preload", R_OK)      = -1
```

- LD有“preload”功能
  - 演示：Hacking with LD_PRELOAD

操作系统眼中的Hello World

- 加载动态链接库(读出动态链接库、mmap赋予正确权限、映射到正确的位置)

```
open("/etc/ld.so.cache", RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=75899, ...}) = 0
mmap(NULL, 75899, PROT_READ, MAP O__PRIVATE, 3, 0) = 0x7ff4db67e000
close(3)                                    = 0
access("/etc/ld.so.nohwcap", F_OK)          = -1 ENOENT
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff4db0a4000
mprotect(0x7ff4db264000, 2097152, PROT_NONE) = 0
mmap(0x7ff4db464000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1
mmap(0x7ff4db46a000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0)
close(3)                                    = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4db67d000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4db67c000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4db67b000
arch_prctl(ARCH_SET_FS, 0x7ff4db67c700) = 0
mprotect(0x7ff4db464000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)         = 0
mprotect(0x7ff4db693000, 4096, PROT_READ) = 0
munmap(0x7ff4db67e000, 75899)               = 0
```

动态链接库：它到底是什么？

- 用-fPIC选项编译成位置无关代码

- 思考：以下操作如何编译成PIC?
  - 调用某个外部函数func (func位置可能不固定，例如来自另一个库)
  - 访问int x (位置也不固定)

- 在静态链接时，外部函数、外部变量的地址可以确定
  - 所以只需要把call/mov等指令的地址留空，链接时填入
  - 动态链接时也可以重填，但有什么坏处?

函数调用和全局变量访问

- func();

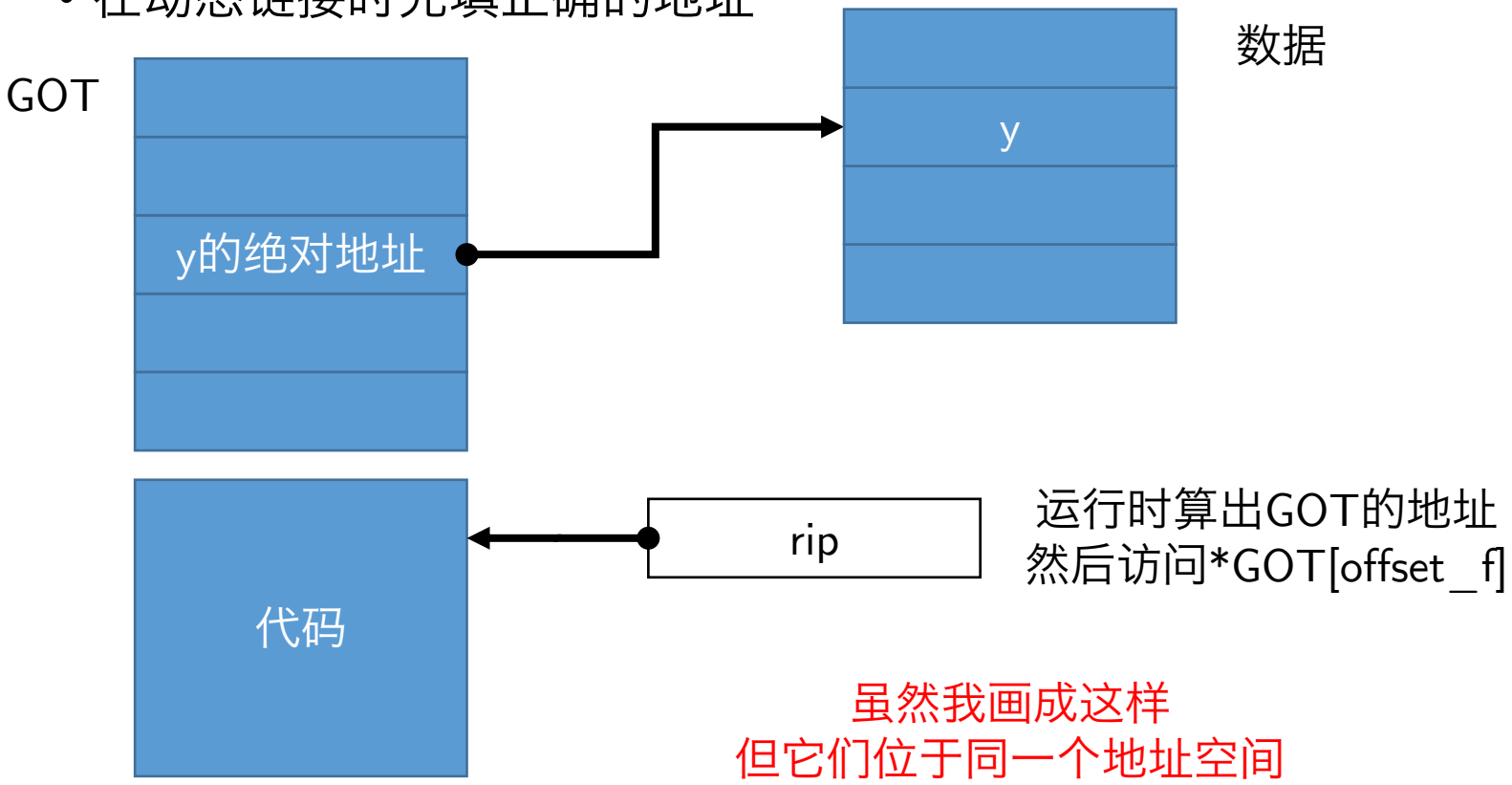
```
(x64) e8 d5 fe ff ff        callq  5a0 <func@plt>
(x32) e8 9c fe ff ff        call   3c0 <func@plt>
```

- y++; // int y;

```
(x64) 48 8b 05 0e 09 20 00 mov    0x20090e(%rip),%rax
(x64) 83 00 01             addl   $0x1,(%rax)
(x32) 8b 83 f4 ff ff ff    mov    -0xc(%ebx),%eax
(x32) 83 00 01             addl   $0x1,(%eax)
```

Global Offset Table (GOT)

- 既然不知道y的地址是什么，就列个表存它的地址
  - 在动态链接时充填正确的地址



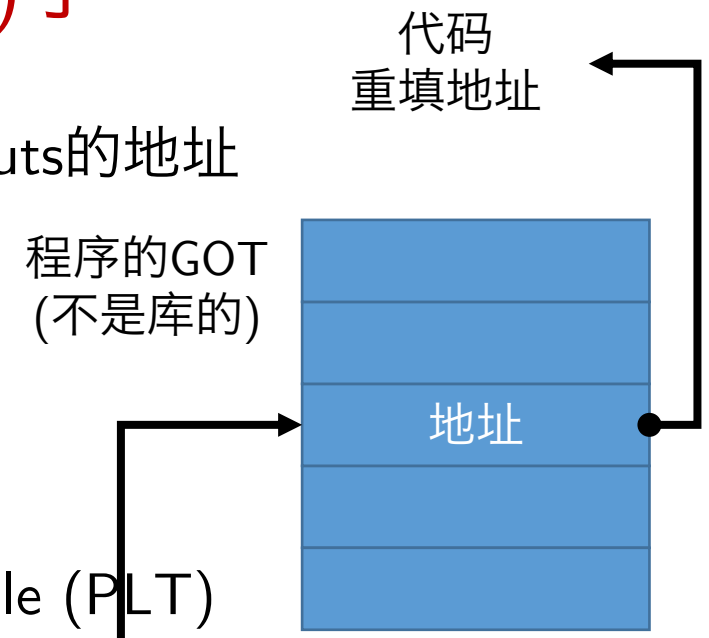
嘿！稍等！

- 如果程序想访问动态链接库里的数据呢？
  - 如果我们直接extern int x;
  - x++会被编译成addl $0x1, 0 (地址等待充填)

- 怎么办？
  - 有个API能根据符号的名字得到动态链接库中的地址
  - 本周的Mini Programming Lab中会用到

终于是时候调用printf (puts)了

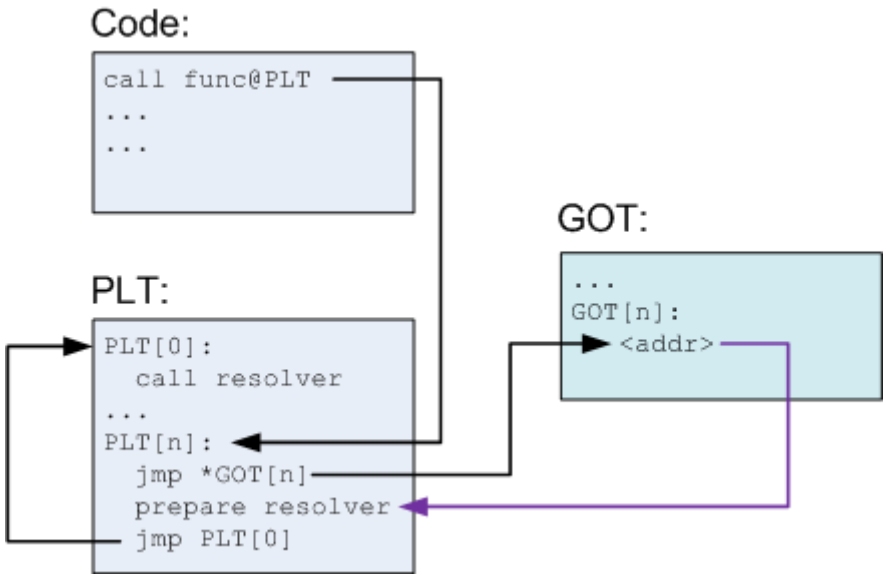
- 在Hello World程序链接时，并不知道puts的地址
  - 只有libc被动态链接时才知道
  - 程序可能有大量库函数调用
    - 如果全部重填相当浪费时间和内存



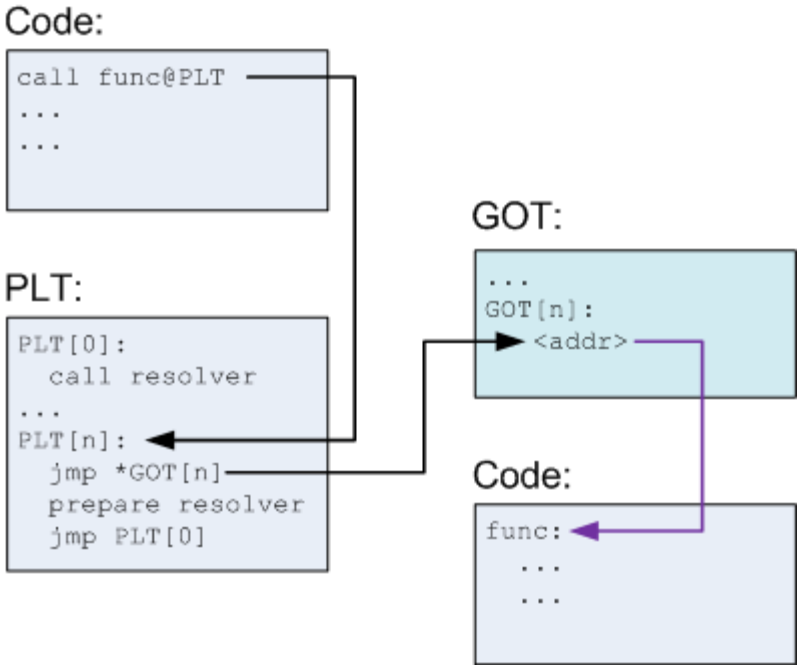
- 解决办法：使用Procedure Linkage Table (PLT)

```
0000000000400400 <puts@plt>:
  400400: ff 25 12 0c 20 00 jmpq   *0x200c12(%rip)
  400406: 68 00 00 00 00    pushq  $0x0
  40040b: e9 e0 ff ff ff    jmpq   4003f0 <_init+0x28>
```

动态链接：过程



首次调用func@plt



后续调用func@plt

Finally, Hello World!

- 终于我们知道printf是如何调用的了
- 之后和静态链接时看到的一样

```
fstat(1, {st_mode=S_IFCHR|0620, ...})   = 0
brk(NULL)                               = 0x1a0d000
brk(0x1a2e000)                          = 0x1a2e000
write(1, "Hello World!\n", 13)           = 13
exit_group(0)                           = ?
```

- Hello, OS World!
- 没想到运行printf竟然如此复杂：理论 ≠ 实践
  - happy hacking!