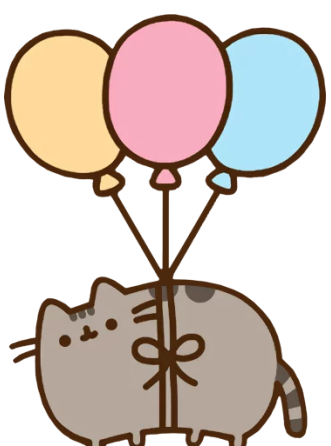
# 写一个不工作的动态链接器

喵喵




对不qǐ
Piàn了大jiā这么久
其是我是小学生
lí jiā出走自己打工
最近实在吃不qǐ饭了
希望大jiā 6.1节
V我50
吃顿好的
xiè xiè gē gē jiě jiě

# 欢度儿童节！

- ELF 基础
- Dynamic linker 基础
- 喵喵 Rant

# 欢度儿童节！

- ELF 基础
- Dynamic linker 基础
- 喵喵 Rant

- 假设听众有一点 ELF 知识基础
- 只处理 Linux 上 x86-64 相关的东西
- 如有问题可以随时打断喵喵！

# Background...

《计算机系统概论》

# Background...

《计算机系统概论》

- https://github.com/CircuitCoder/ld.meow.so
- https://maskray.com
- https://jia.je

# Background...

《计算机系统概论》

- https://github.com/CircuitCoder/ld.meow.so
- https://maskray.com
- https://jia.je

- Load ELF
- Link ELF
- ~~???~~
- PROFIT

# What could possibly go wrong

# Basics

PIE: Position-independent executable

可以被放置在任意基地址被执行

# Basics

PIE: Position-independent executable

可以被放置在任意基地址被执行

- `0x114514(%rip)` on x86
- `auipc` on RISC-V

# Basics

将链接过程“延迟”到运行时

```
int meow(int);
meow(1);
```

```
int (*meow)(int);
meow = ...;
meow(1);
```

# Basics

将链接过程“延迟”到运行时

```
int meow(int);
meow(1);
```

```
int (*meow@got)(int);
meow@got = ...;
meow@got(1);
```

# Basics

- `.so` / Shared objects = 编译单元

# Basics

- `.so` / Shared objects = 编译单元
- `.dynamic` 段中包含各种需要的表的地址
  - `DT_NEEDED`: 依赖的 `.so`
  - `DT_SYMTAB`: 符号表
  - Relocation tables

# Basics

- `.so` / Shared objects = 编译单元
- `.dynamic` 段中包含各种需要的表的地址
  - `DT_NEEDED`: 依赖的 `.so`
  - `DT_SYMTAB`: 符号表
  - Relocation tables
- Kernel 根据 `PT_INTERP` 程序头选择动态链接器

# PT_INTERP...?

它真的是 Interpreter

```
$ /usr/lib/ld-linux-x86-64.so.2 /usr/bin/ls
```

# PT_INTERP...?

它真的是 Interpreter

```
$ /usr/lib/ld-linux-x86-64.so.2 /usr/bin/ls
```

- Loader
- Linker

# PT_INTERP...?

它真的是 Interpreter

```
$ /usr/lib/ld-linux-x86-64.so.2 /usr/bin/ls
```

- Loader
- Linker
- (Part of) runtime

# Optimization, Self-relocation & Zig

Folklore: “ld.so 不能开 O2 编译”

```
# ifdef HAVE_BUILTIN_MEMSET
  __builtin_memset (bootstrap_map.l_info, '\0', sizeof
(bootstrap_map.l_info));
# else
  for (size_t cnt = 0; cnt < len; ++cnt)
    bootstrap_map.l_info[cnt] = 0;
# endif
```

glibc/elf/rtld.c: <_dl_start>

```c
int meow(int *output, int len) {
  for(int i = 0; i < len; ++i) output[i] = 0;
}
```

```
0000000000001150 <meow>:
  ...
  1164: e8 c7 fe ff ff          call   1030 <memset@plt>
  ...
```

# Self-relocation

动态链接器链接所有不是动态链接器的动态程序，请问：谁动态链接动态链接器?

Dynamic linker dynamically links all dynamic programs that is not a dynamic linker, who dynamically links the dynamic linker?

# Self-relocation

动态链接器链接所有不是动态链接器的动态程序，请问：谁动态链接动态链接器?

Dynamic linker dynamically links all dynamic programs that is not a dynamic linker, who dynamically links the dynamic linker?

Self-relocation

# 加载顺序

- ld.so
- LD_PRELOAD
- libc.so
- application

# 加载顺序

- ld.so
- LD_PRELOAD
- libc.so
- application

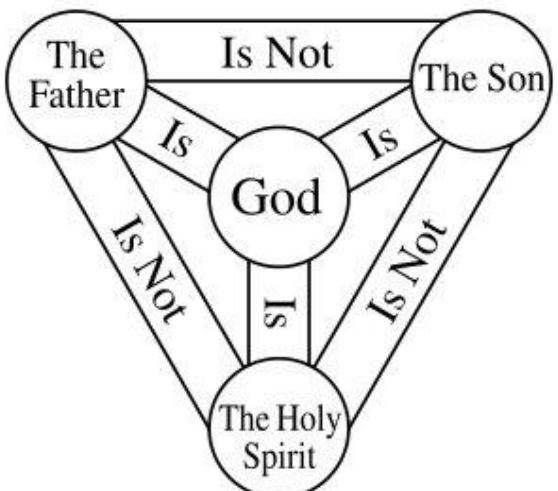
Also somewhere: vDSO

```
# On targets without __builtin_memset, rtld.c uses a hand-
coded loop
# in _dl_start.  Make sure this isn't turned into a call to
regular memset.
ifeq (yes,$(have-loop-to-function))
CFLAGS-rtld.c += -fno-tree-loop-distribute-patterns
endif
```

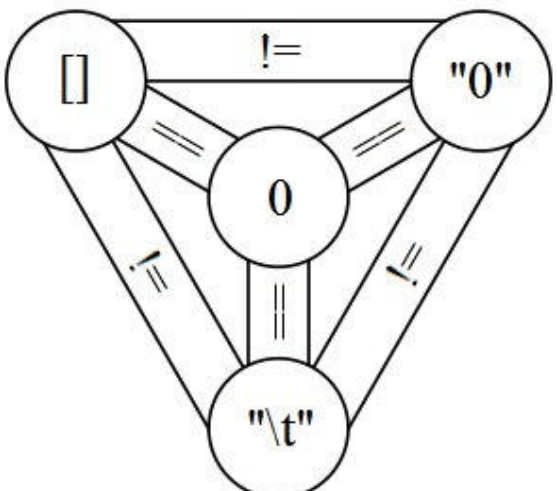
glibc/elf/Makefile

# What about MUSL?

# What about MUSL?



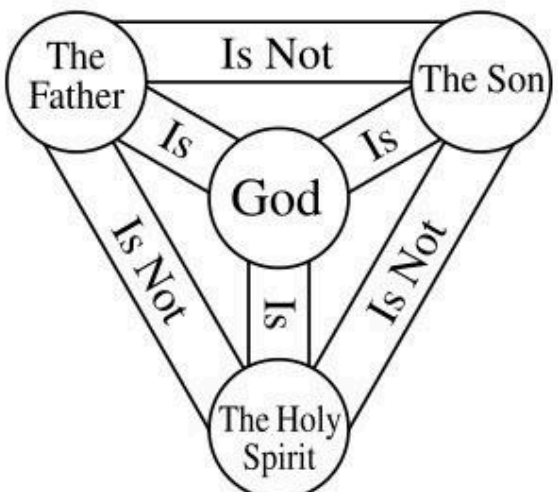

Christianity



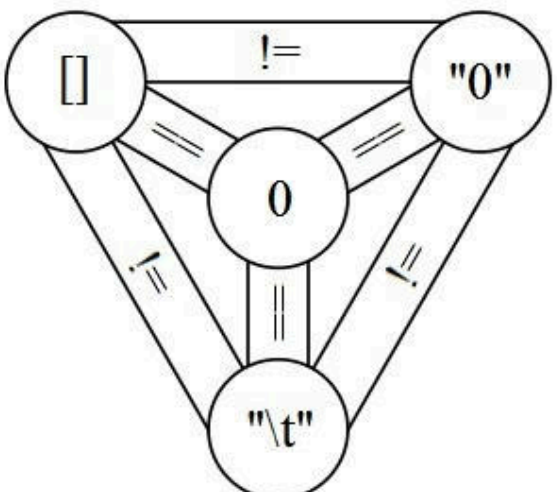

JavaScript

@hsjoihs

# What about MUSL?

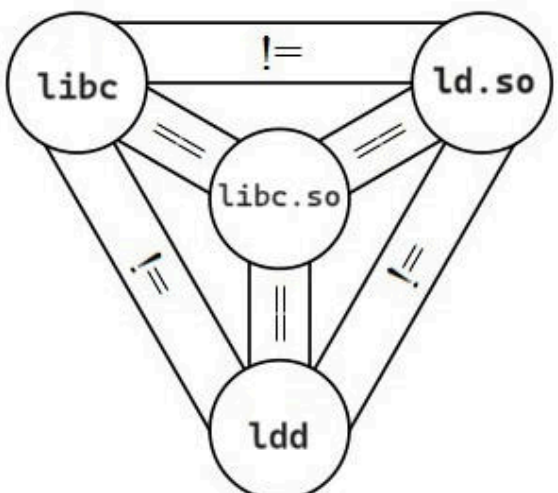



Christianity




JavaScript

@hsjoihs




MUSL Project

# What about MUSL?

```
➜ linker ls -ll /lib/ld-musl-x86_64.so.1
lrwxrwxrwx 1 root root 25 Mar  2 19:37 /lib/ld-musl-x86_64.so.1
 -> /usr/lib/musl/lib/libc.so
➜ linker ln -s /lib/ld-musl-x86_64.so.1 /tmp/ldd
➜ linker /tmp/ldd /tmp/ldd
        /tmp/ldd (0x7fbade552000)
```

# What about MUSL?

```
➜ linker ls -ll /lib/ld-musl-x86_64.so.1
lrwxrwxrwx 1 root root 25 Mar  2 19:37 /lib/ld-musl-x86_64.so.1
 -> /usr/lib/musl/lib/libc.so
➜ linker ln -s /lib/ld-musl-x86_64.so.1 /tmp/ldd
➜ linker /tmp/ldd /tmp/ldd
        /tmp/ldd (0x7fbade552000)
```

`glibc ld.so` + `musl libc` 会爆炸

# What about MUSL?

```
➜ linker ls -ll /lib/ld-musl-x86_64.so.1
lrwxrwxrwx 1 root root 25 Mar  2 19:37 /lib/ld-musl-x86_64.so.1
 -> /usr/lib/musl/lib/libc.so
➜ linker ln -s /lib/ld-musl-x86_64.so.1 /tmp/ldd
➜ linker /tmp/ldd /tmp/ldd
        /tmp/ldd (0x7fbade552000)
```

glibc ld.so + musl libc 会爆炸

glibc ldd + musl binary 会爆炸

# What about MUSL?

```
➜ linker ls -ll /lib/ld-musl-x86_64.so.1
lrwxrwxrwx 1 root root 25 Mar  2 19:37 /lib/ld-musl-x86_64.so.1
 -> /usr/lib/musl/lib/libc.so
➜ linker ln -s /lib/ld-musl-x86_64.so.1 /tmp/ldd
➜ linker /tmp/ldd /tmp/ldd
        /tmp/ldd (0x7fbade552000)
```

glibc ld.so + musl libc 会爆炸

glibc ldd + musl binary 会爆炸

musl ldd + glibc libc?

```
if (find_sym(&temp_dso, "__libc_start_main", 1).sym &&
  find_sym(&temp_dso, "stdin", 1).sym) {
    unmap_library(&temp_dso);
    return load_library("libc.so", needed_by);
}
```

musl/ldso/dynlink.c

# Zig

# Zig

~~纯粹是想玩 Zig~~

# Zig

~~纯粹是想玩 Zig~~

“Free-standing”: 并不依赖 `libc`，标准库直接通过 `syscall` 实现。

# Zig

~~纯粹是想玩 Zig~~

“Free-standing”: 并不依赖 `libc`，标准库直接通过 `syscall` 实现。

- ```
  fn meow();
  meow();
  ```
  ```

# Zig

~~纯粹是想玩 Zig~~

“Free-standing”: 并不依赖 `libc`，标准库直接通过 `syscall` 实现。

- ```
  fn meow();
  meow();
  ```
- ```
  var meow: u64 = 0;
  ```

# Zig

~~纯粹是想玩 Zig~~

“Free-standing”: 并不依赖 `libc`，标准库直接通过 `syscall` 实现。

- ```
  fn meow();
  meow();
  ```
- ```
  var meow: u64 = 0;
  ```
- ```
  const meow: [*]u8 = "Meow-meow";
  ```
  ```

# So far...

- 链接器自己在内存里
- 程序不知道在哪儿
- 需要链接自己
- 需要加载依赖
- 需要链接程序

# So far...

- 链接器自己在内存里
- 程序不知道在哪儿
- 需要链接自己
- 需要加载依赖
- 需要链接程序

Next up: Kernel 给了我们什么?

# AUX vector

内核栈的最顶端:

```
{ argc, argv, envp, aux }
```

# AUX vector

内核栈的最顶端:

```
{ argc, argv, envp, aux }
```

```
struct aux_t {
  size_t a_type;
  size_t a_val;
}
```

# AUX vector

内核栈的最顶端：

```
{ argc, argv, envp, aux }
```

- `AT_BASE`: Interpreter 加载基址
- `AT_PHDR`: 用户程序 Program header 基址
- `AT_EXECFN`: 用户程序路径
- `AT_EXECFN_SYSINFO_EHDR`: vDSO ELF header 地址

# AUX vector

内核栈的最顶端:

```
{ argc, argv, envp, aux }
```

- `AT_BASE`: Interpreter 加载基址
- `AT_PHDR`: 用户程序 Program header 基址
- `AT_EXECFN`: 用户程序路径
- `AT_EXECFN_SYSINFO_EHDR`: vDSO ELF header 地址

```
$ /usr/lib/ld-linux-x86-64.so.2 /usr/bin/ls
```

- 基址: `aux AT_BASE` 或者 `__ehdr_start`
- 用户程序可能需要自己加载。用户程序路径可能是 Interpreter 自己

- 基址: `aux AT_BASE` 或者 `__ehdr_start`
- 用户程序可能需要自己加载。用户程序路径可能是 Interpreter 自己
- `_DYNAMIC` 符号指向 `.dynamic` 段开始

- 基址: `aux AT_BASE` 或者 `__ehdr_start`
- 用户程序可能需要自己加载。用户程序路径可能是 Interpreter 自己
- `_DYNAMIC` 符号指向 `.dynamic` 段开始

根据 `_DYNAMIC` 和基址可以完成 Self-relocations

# 加载用户程序

PT_LOAD: 加载一块儿 ELF 的内容到内存里（Segment）

# 加载用户程序

`PT_LOAD`: 加载一块儿 ELF 的内容到内存里（Segment）

- `offset`, vaddr
- file size, mem size
- flags

# 加载用户程序

`PT_LOAD`: 加载一块儿 ELF 的内容到内存里（Segment）

- `offset`, vaddr
- file size, mem size
- flags

所有这些值都不一定是页对齐的

# 对齐

极端情况：

- `offset` 不对齐：映射出来的内容前面有垃圾
- `mem size` 不对齐：映射出来的内容后面有垃圾

# 对齐

极端情况：

- `offset` 不对齐：映射出来的内容前面有垃圾
- `mem size` 不对齐：映射出来的内容后面有垃圾
- `mem size > file size`：需要两次 `mmap`：一次基于文件，一次 anonymous，否则 SIG🚌

# 对齐

极端情况:

- `offset` 不对齐：映射出来的内容前面有垃圾
- `mem size` 不对齐：映射出来的内容后面有垃圾
- `mem size > file size`：需要两次 `mmap`: 一次基于文件，一次 anonymous，否则 SIG🚌
- 可读写: 需要清空尾巴上的内容。`(.bss)`

# 对齐

极端情况：

- `offset` 不对齐：映射出来的内容前面有垃圾
- `mem size` 不对齐：映射出来的内容后面有垃圾
- `mem size > file size`：需要两次 `mmap`：一次基于文件，一次 anonymous，否则 SIG🚌
- 可读写：需要清空尾巴上的内容。`(.bss)`

只读情况呢？前面的垃圾呢？

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_1)$

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_1)$
- `mmap` $[b_2, e_2)$
- ...

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_1)$
- `mmap` $[b_2, e_2)$
- ...

```
static int meow = 0;
meow = 1;
```

⇓

```
mov $0x0, 0x114514(%rip)
```

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_1)$，得到基址
- `mmap` $[b_2, e_2)$，使用 `MAP_FIX_NOREPLACE`
- ...

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_1)$，得到基址
- `mmap` $[b_2, e_2)$，使用 `MAP_FIX_NOREPLACE`
- ...

`EEXIST`

# One more thing...

$[b_1, e_1), [b_2, e_2), ..., [b_n, e_n)$

- `mmap` $[b_1, e_n)$，得到基址
- **`munmap`** $[e_1, e_n)$
- `mmap` $[b_2, e_2)$，使用 `MAP_FIX_NOREPLACE`
- ...

# It works!

如果是简单的 `a.out` 依赖 `b.so`，`b.so` 无依赖，现在应该可以直接执行了！

直到尝试执行一个依赖 libc 的程序...

# It works!

如果是简单的 `a.out` 依赖 `b.so`，`b.so` 无依赖，现在应该可以直接执行了！

直到尝试执行一个依赖 libc 的程序...

`libc.so` is very special

# Meanwhile...

```
class VeryInnocentClass {
  VeryInnocentClass() {
    prints("Hi");
  }
  ~VeryInnocentClass() {
    prints("Bye");
  }
}
```

# Meanwhile...

```
class VeryInnocentClass {
  VeryInnocentClass() {
    prints("Hi");
  }
  ~VeryInnocentClass() {
    prints("Bye");
  }
}

static VeryInnocentClass meow;
```

# DT_INIT / DT_FINI

Also DT_INIT_ARRAY, DT_FINI_ARRAY

# DT_INIT / DT_FINI

Also DT_INIT_ARRAY, DT_FINI_ARRAY

- INIT 在转移给用户程序之前执行
- FINI 需要“保证在退出的时候执行”

# DT_INIT / DT_FINI

Also DT_INIT_ARRAY, DT_FINI_ARRAY

- INIT 在转移给用户程序之前执行
- FINI 需要“保证在退出的时候执行”

`atexit`, 和 libc 耦合

# DT_INIT / DT_FINI

Also DT_INIT_ARRAY, DT_FINI_ARRAY

- INIT 在转移给用户程序之前执行
- FINI 需要“保证在退出的时候执行”

`atexit`, 和 libc 耦合

- `__attribute__((constructor))`
- `__attribute__((destructor))`

# Itanium ABI

https://itanium-cxx-abi.github.io/cxx-abi/abi.html#dso-dtor

```
extern "C" int __cxa_atexit ( void (*f)(void *), void *p,
void *d );
```

Destructor 在 `.init / .init_array` 中注册。

# Itanium ABI

https://itanium-cxx-abi.github.io/cxx-abi/abi.html#dso-dtor

```
extern "C" int __cxa_atexit ( void (*f)(void *), void *p,
void *d );
```

Destructor 在 `.init / .init_array` 中注册。

```
void* __dso_handle = &handle;
```

# Finally...

可以执行绝大多数 C++ 的代码了...

调用 libc?

# Finally...

可以执行绝大多数 C++ 的代码了...

调用 libc?

```
thread_local int errno;
```

# Thread-local storage

# Thread-local storage

Thread control block + TLS

# Thread-local storage

Thread control block + TLS

- TLS 中的内容需要特殊的链接：TPOFF，以及一个函数 `__tls_get_addr`
- `pthread_create()` 时，需要新分配 TLS 空间：需要 ld.so 配合。
- TLS 局部状态保存在 ld.so 初始化时的地址空间中：interp 和 libc 不能互相交叉使用。

```
R_X86_64_GLOB_DAT _dl_argv@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_find_dso_for_[...]@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_deallocate_tls@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_signal_error@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_signal_exception@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_audit_symbind_alt@GLIBC_PRIVATE + 0
R_X86_64_TPOFF64  __libc_dlerror_result@@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_rtld_di_serinfo@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_allocate_tls@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_catch_exception@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_allocate_tls_init@GLIBC_PRIVATE + 0
R_X86_64_GLOB_DAT _dl_audit_preinit@GLIBC_PRIVATE + 0
```

# Conclusion

为了写一个工作的 ELF Dynamic linker，你需要：

# Conclusion

为了写一个工作的 ELF Dynamic linker，你需要：

- 实现一个 libc

# Conclusion

为了写一个工作的 ELF Dynamic linker，你需要：

- 实现一个 libc
- 实现一个 pthread

# Conclusion

为了写一个工作的 ELF Dynamic linker，你需要：

- 实现一个 libc
- 实现一个 pthread
- 在编译器打好后门

# Conclusion

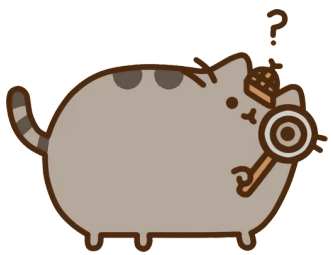
为了写一个工作的 ELF Dynamic linker，你需要：

- 实现一个 libc
- 实现一个 pthread
- 在编译器打好后门

一个不工作的 ELF Dynamic linker 可以实现的是：

- 支持 Free-standing C
- 差不多支持 Free-standing C++

# Question time!



https://github.com/CircuitCoder/ld.meow.so