# 你所不知道的 C 語言：連結器和執行檔資訊



直播錄影

## 簡介

連結器 (linker) 是很多 C 程式開發者會忽略的議題，但在 Linux 核心和 Android Open Source Project (AOSP) 這樣包含大量 C 程式的專案中，不難見到連結器的身影，像是客製化的 linker script, 針對 gnu ld / gold linker 的最佳化，甚至是搭配編譯器和連結器選項，提供可掛載的核心模組 (Linux kernel module) 功能。

本講座先回顧在 1970 年 UNIX 剛被創造出來時，系統提供的 loader 是如何演化為 ld (現在的 UNIX 世界的 linker)，及其名稱暗示著程式載入器的作用，述及 gcc 和連結器相關的 GNU extension 可如何運用 (如嵌入一份圖片內容到執行檔中)，搭配分析工具探討 ELF 執行檔和連結器的交叉運作機制。

另外，慶祝高雄愛河正名為「金銀河」，本講座要探討 gold (別懷疑，真的有一個 linker 名稱就叫做「金」) 如何讓 Linux 核心發揮 Link-Time Optimization (LTO) 效益，編譯出更精簡且更高效的 Linux 核心映像檔。

## 嵌入一個二進位檔案到執行檔

假設有個二進位檔案名為 `blob`，可善用 `xxd` 工具:

```
$ xxd --include blob
```

為了解說方便，先製造一個檔案，紀錄長度:

```
$ uname -a > blob
$ uname -a | wc -c
105
```

將 `blob` 納入 ELF 檔案中:

```
$ ld -s -r -b binary -o blob.o blob
```

觀察產生的 `blob.o`:

```
$ objdump -t blob.o

blob.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .data 0000000000000000 .data
0000000000000069 g       .data 0000000000000000 _binary_blob_end
0000000000000000 g       .data 0000000000000000 _binary_blob_start
0000000000000069 g       *ABS* 0000000000000000 _binary_blob_size
```

寫個測試程式 ( `test.c` ):

```c
#include <stdio.h>
int main(void) {
    extern void *_binary_blob_start, *_binary_blob_end;
    void *start = &_binary_blob_start, *end = &_binary_blob_end;
    printf("Data: %p..%p (%zu bytes)\n",
           start, end, end - start);
    return 0;
}
```

編譯、連結，和執行:

```
$ gcc test.c blob.o -o test
$ ./test
Data: 0x55ed5ed15010..0x55ed5ed15079 (105 bytes)
```

對照上面的 `105` bytes，符合。

回頭看稍早產生的 `blob.o`:

```
$ readelf -S blob.o
There are 5 section headers, starting at offset 0x180:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS         0000000000000000  00000040
       0000000000000069  0000000000000000  WA       0     0     1
  [ 2] .symtab           SYMTAB           0000000000000000  000000b0
       0000000000000078  0000000000000018           3     2     8
```

```
  [ 3] .strtab            STRTAB           0000000000000000  00000128
       0000000000000037  0000000000000000           0     0     1
  [ 4] .shstrtab          STRTAB           0000000000000000  0000015f
       0000000000000021  0000000000000000           0     0     1
```

另一個示範 objcopy_to_carray

## init script 示範

在 F9 microkernel 有個特徵 Init hooks，允許特定程式碼在核心啟動早期就執行。使用方式:

```
#include <init_hook.h>
#include <debug.h>

void hook_test(void) {
    dbg_printf(DL_EMERG, "hook test\n");
}
INIT_HOOK(hook_test, INIT_LEVEL_PLATFORM - 1)
```

透過 GNU extension 去指定 ELF section: include/init_hook.h:

```
#define INIT_HOOK(_hook, _level) \
    const init_struct _init_struct_##_hook \
            __attribute__((section(".init_hook"))) = { \
        .level = _level, \
        .hook = _hook, \
        .hook_name = #_hook, \
    };
```

在 platform/stm32f4/f9.ld 配置了 `.init_hook` 的空間:

```
                init_hook_start = .;
                KEEP(*(.init_hook))
                init_hook_end = .;
```

最後在 kernel/init.c 就清晰了:

```
extern const init_struct init_hook_start[];
extern const init_struct init_hook_end[];
static unsigned int last_level = 0;

int run_init_hook(unsigned int level) {
    unsigned int max_called_level = last_level;

    for (const init_struct *ptr = init_hook_start;
         ptr != init_hook_end; ++ptr)
        if ((ptr->level > last_level) &&
            (ptr->level <= level)) {
            max_called_level = MAX(max_called_level, ptr->level);
            ptr->hook();
    }

    last_level = max_called_level;
    return last_level;
}
```

## 連結器在軟體最佳化扮演重要角色

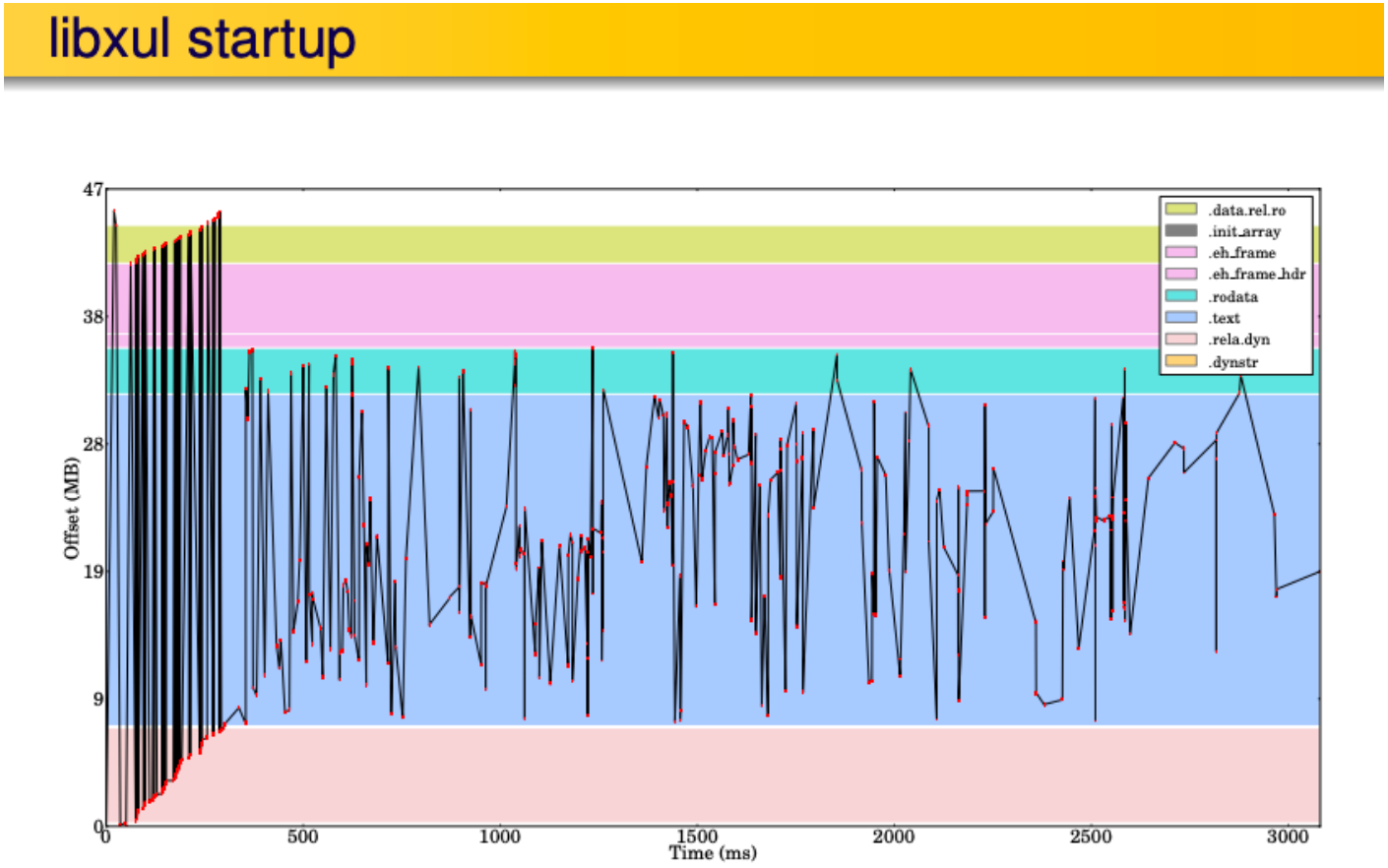
複習「你所不知道的 C 語言」: 編譯器和最佳化原理篇 動態連結器篇

Ian Wienand 的電子書: Chapter 7. The Toolchain
對照閱讀:

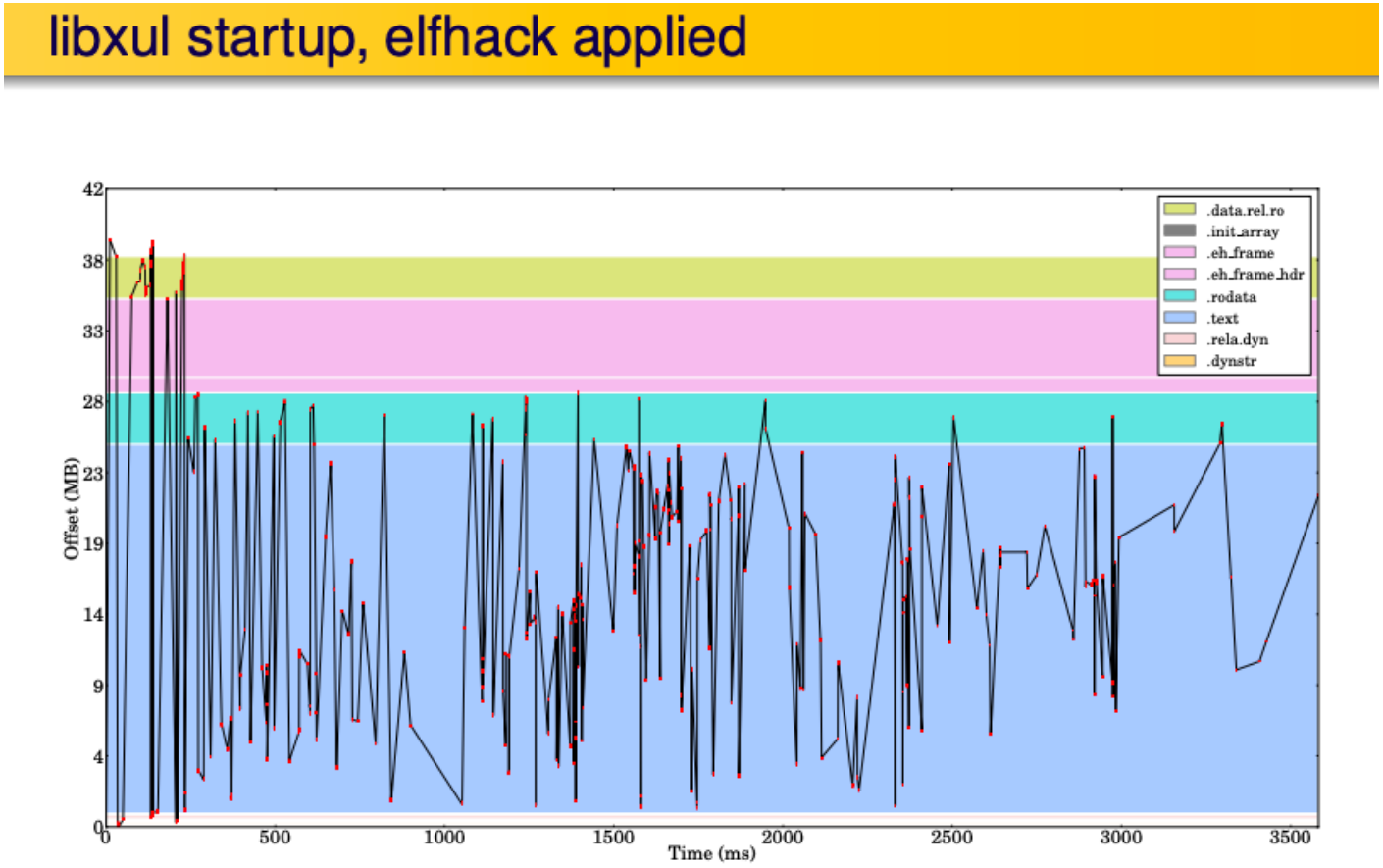
- 10分鐘讀懂 linker scripts
- Linker Script 初探 - GNU Linker ld 手冊略讀

Optimizing large applications (2013)

- 原本的執行路徑



- elfhack 調整後




  - 20% of Firefox libxul image are relocations
  - ELF relocations are not terribly size optimized
    - REL relocations on x86 take 8 bytes
    - RELA relocation on x86-64 take 24 bytes
  - Elfhack compress the relocations
    - ELFhack removes IP relative ELF relocations and store them in compact custom format. It handles well sequences of IP relative relocations in vtables.

        - After ELF linking, ELFhack linking completes the process.
        - ELFhack is general tool but not compatible with -z relro security feature.
    - `7.5` MB of relocations → `0.3` MB.
    - 搭配閱讀: Improving libxul startup I/O by hacking the ELF format

- Linktime optimization in GCC (2014)

    - part 1 - brief history
    - part 2 - Firefox
    - part 3 - LibreOffice

What makes LLD so fast?

- WebM 錄影
- Peter Smith 宣稱 lld 比 GNU gold linker 快 2 到 3 倍，又比標準的 `ld.bfd` 快 5 到 10 倍

The missing link: explaining ELF static linking, semantically

> In the C programming language, a simple program such as ‘hello, world!’ exercises very few features of the language, and can be compiled even by a toy compiler. However, for a linker, even the smallest C program amounts to a complex job, since it links with the C library—one of the most complex libraries on the system, in terms of the linker features it exercises.

## 在 Linux 核心的應用案例

Shrinking the kernel with link-time garbage collection

Shrinking the kernel with link-time optimization

- Dead-code elimination
- LTO and the kernel

Shrinking the kernel with an axe

## 其他

- tramp-test

```
$ ./dis.sh
$ cat tramp_test.o.asm
$ cat trampoline.o.asm
```

- libelfmaster: Secure ELF parsing/loading library for forensics reconstruction of malware, and robust reverse engineering tools
- dt_infect: ELF Shared library injector using DT_NEEDED precedence infection. Acts as a permanent LD_PRELOAD
- How the GNU C Library handles backward compatibility