C++

Spring 2021

compscicenter.ru

Грабовой Филипп, t.me/phil-grab

Lecture VI

Linking, static & dynamic libraries, loader

Linking

symbols

- code w/ using symbols
- code w/ symbols definition
- цель линковки — заменить символы/лейблы на конкретные адреса
  - глобальные переменные ⇒ адреса ячеек с их значениями
  - вызовы функций ⇒ адреса для выполнения jmp
  - и т.д.

ld inputs

- object files (ELF/COFF/...)
- внутри: sections + headers
- некоторые секции и их содержимое:
  - symbol table + relocation records
    - `.symtab` + `.rela.<section>`
  - debugging info
  - dynamic linking information

sections + process image

- еще примеры секций: .text, .bss, .data, .rodata
- code w/ definitions
- некоторые секции будут использованы в process image напрямую
  - как страницы памяти

relocation

- процесс подстановки нужных адресов вместо символов
- секции с использованиями: `.rela.data`, `.rela.text`
- секции с определениями (и адресами): `.symtab`
  - *посмотрим objdump*

linking statically

- релокация происходит во время линковки
  - результат — единый файл
- можно линковаться с уже готовой *статической* библиотекой
  - файлы `libxxx.a` в Linux, `xxx.lib` в Windows

linking dynamically

- релокация отложена (на время исполнения)
  - результат — в exec появляется информация *о зависимости*
  - это файлы `libxxx.so` в Linux, `xxx.dll` в Windows
  - в них — необходимые определения + информация о динамической линковке
- символы надо будет *дереферить* в рантайме
  - на старте программы или позже

link-time optimizations [*]

- ограничения оптимизаций при компиляции — внешние интефейсы (boundaries)
- идея: оптимизировать и их, на этапе линковки
  - объектные файлы `.o`: машинный код или Intermediate Language (IL)
  - + применение бекенда-оптимизатора
- MSVC: Link Time Code Generation, details
- LLVM: Link Time Optimization (libLTO), doc with example

static & dynamic libraries

static library

- архив с объектными файлами
- линкуется с программой до запуска
- при запуске разных программ одинаковые либы не переиспользуются
  - в рантайме — дублирование данных

dynamic library

- формат — похож на исполняемый файл
- подтягивается во время исполнения
  - есть интерфейс для отложенной/ленивой загрузки (плагины)
- в рантайме по возможности переиспользуется между процессами
  - код из библиотеки — переиспользуется, данные — свои (странички создаются CoW)

static vs dynamic

- `s > d`: код влинковывается ⇒ нет проблем с зависимостями в рантайме
- `s > d`: попадает только необходимый библиотечный код (с оптимизациями[*])
- `d > s`: размер файла/process image
- `d > s`: возможность прозрачно заменить зависимость в среде исполнения

static library creating

- MSVS: изменить тип таргета (Static Library, .lib)
- Linux:

```
$ g++ -Wall -c *.cpp
$ ar -cv libsome.a *.o
```

dynamic library creating

- MSVS: изменить тип таргета (Dynamic Library, .dll)
- Linux:

```
$ g++ -Wall -c *.cpp
$ g++ -shared -o libsome.so *.o
```

exporting symbols (dynamic library)

- В MSVC по-умолчанию никакие — надо перечислять:
  - при компиляции
  - Module Definition Files
  - `__declspec(dllexport)`
  - ...
- В GCC — все
  - → можно скрывать `__attribute__((visibility("hidden")))`

explicit linking & usage

- MSVC:
  - LoadLibrary*, GetProcAddress

```
typedef void(*func_pointer)(int, double);

HMODULE lib = LoadLibrary("some.dll"); // or LoadLibrary
auto func = reinterpret_cast<func_pointer>(GetProcAddres

FreeLibrary(lib);
```

- Linux: `dlopen`, `dlsym`, `dlclose`
- есть хуки на загрузку/выгрузку библиотек (*)

implicit linking

- компиляция → `.lib` + `.dll`. Статическая часть:
  - секции, задающие dynamic-зависимости
  - код оберток, проксирующих вызовы (зачастую оптимизируется)

notes

- если нет уверенности в совпадении ABI:
  - функции стоит помечать `extern "C"` и фиксировать calling convention
  - перегрузки функций, методы классов, обработка исключений — могут не работать
    - ⇒ стоит избегать фичей плюсового рантайма
- если и исполняемый файл и динамическая библиотека собираются в одном проекте — одинаковые компилятор/флаги сборки/платформа, — то совпадение ABI у вас есть

Loaders by example

example

- пример: `printf(…)` — функция, использующая libc
- *посмотрим на* `$ g++ test.cpp && sudo strace ./a.out`
  - *код в* `./src`
  - *команды в* `Makefile`

steps

- загрузить код приложения в память (`exec*` sys calls)
- приготовить динамические зависимости (информация — из секций):
  - загрузить Shared Objects в память (рекурсивно с зависимостями)
  - выполнить релокации (→ Global Offset Table, Procedure Linkage Table)
  - инициализировать окружение зависимости (entry point)
- инициализировать окружение
- передать выполнение в main

entry points

- code example
  - implicit init on startup (constructor)
  - implicit teardown on exit (calling destructor as exit)
- есть способы переопределять точки входа в DLL:
  - в Windows: `DllMain`
  - в Linux: `__attribute__((constructor))`, `__attribute__((destructor))`

links

- CSC: Низкоуровневый взгляд на динамические библиотеки и модели кода
- CppCon 2017: James McNellis “Everything You Ever Wanted to Know about DLLs”