Динамические библиотеки и способы их ускорения

C++ Russia 2024

Обо мне

- Юрий Грибов (yugr)
- Инженер-компиляторщик
- Gmail: tetra2005
- t.me/the_real_yugr
- https://github.com/yugr
- https://www.linkedin.com/in/yugr/

План доклада

- Динамические библиотеки

План доклада

- Динамические библиотеки
  - Отличия от статических библиотек
  - Принципы работы
  - Преимущества и недостатки

План доклада

- Динамические библиотеки
  - Отличия от статических библиотек
  - Принципы работы
  - Преимущества и недостатки
- Сравнение реализаций в Linux и Windows

План доклада

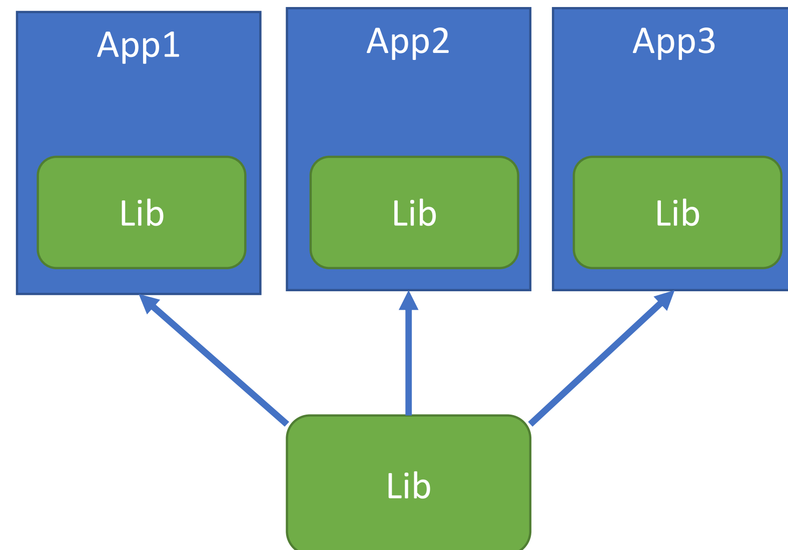
- Динамические библиотеки
  - Отличия от статических библиотек
  - Принципы работы
  - Преимущества и недостатки
- Сравнение реализаций в Linux и Windows
- Ускорение работы динамических библиотек

План доклада

- Динамические библиотеки
  - Отличия от статических библиотек
  - Принципы работы
  - Преимущества и недостатки
- Сравнение реализаций в Linux и Windows
- Ускорение работы динамических библиотек
  - Причины накладных расходов
  - Способы их уменьшения в современных тулчейнах

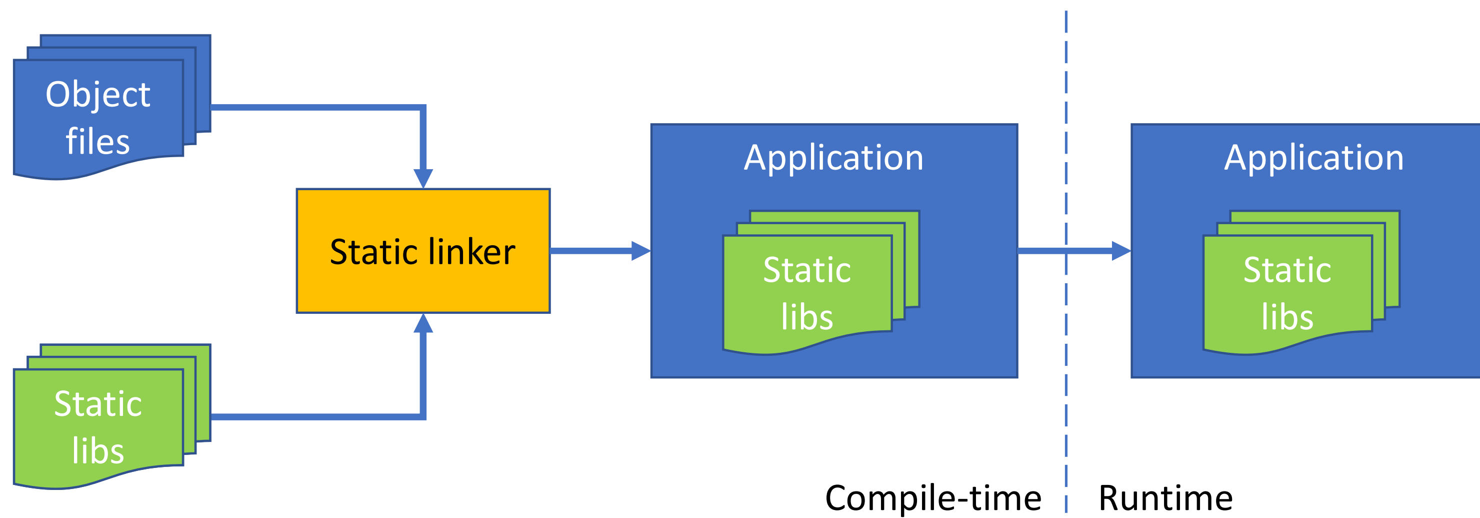
Библиотеки

- Архивы переиспользуемого кода
- Могут быть использованы в нескольких программах
- В зависимости от времени связывания (link time) могут быть
  - Статическими (.a, .lib)
  - Динамическими (.so, .dll, .dylib)
- Операционные системы поддерживают оба вида библиотек
  - Windows, Linux, macOS, BSDs



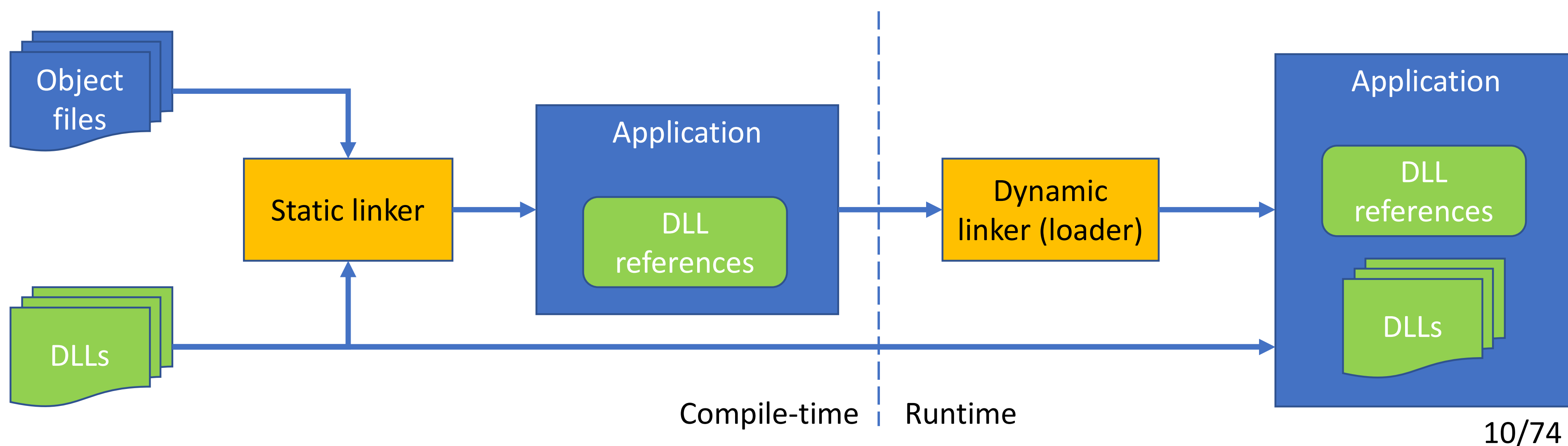
Статические библиотеки

- Static libraries
- Становятся частью исполняемого файла программы на этапе линковки



Динамические библиотеки

- Dynamic-link libraries (DLL), shared libraries, shared objects
- Не являются частью исполняемого файла программы
- Обычно загружаются в начале работы программы



Использование динамических библиотек

- Два основных способа:
  - Традиционное link-time связывание

```
gcc program.o -lgmp
link.exe program.obj libgmp.lib
```

  - Связывание на этапе исполнения (run-time loading, dynamic loading)

```
void *lib = dlopen("libgmp.so", RTLD_LAZY | RTLD_GLOBAL);
HANDLE lib = LoadLibrary("libgmp.dll");
```

- При традиционном связывании библиотека будет загружена на старте программы
- При втором варианте – в любом месте программы
  - Открывает возможность для использования lazy loading, плагинов и пр.

Преимущества DLL

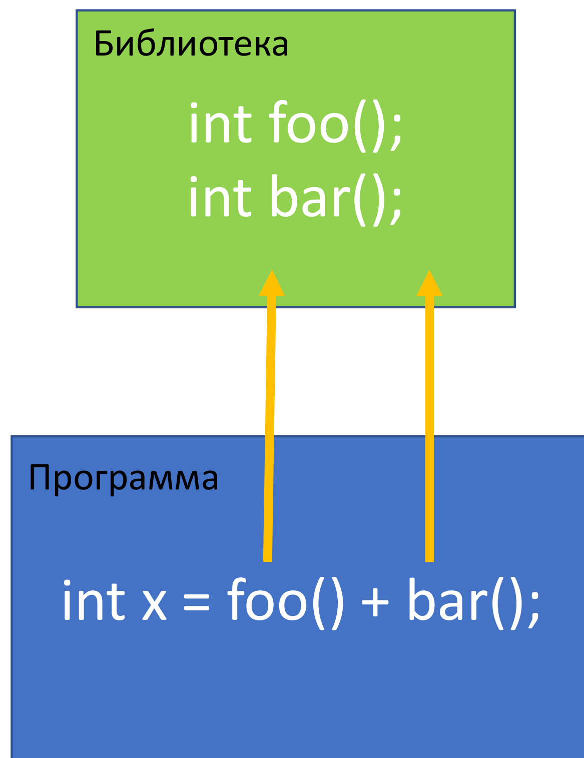
- Экономия оперативной памяти и диска
  - ~1.1G RAM на Ubuntu Desktop[1,2] (с запущенными Firefox/KOffice/Thunderbird)
  - ~10G HDD на Ubuntu Desktop[1] (с Firefox, KOffice, etc.)
- Быстрые системные обновления
  - Зависимые файлы не нужно перекомпилировать при обновлении библиотеки
- Поддержка более сложных сценариев работы:
  - Отложенная загрузка (lazy loading)
  - Загрузка пользовательских динамических расширений (плагинов)
  - Загрузка наиболее различных версий библиотеки в зависимости от окружения (например от возможностей процессора)

1) Детали всех замеров приведены в приложении к презентации https://github.com/yugr/CppRussia/tree/master/2024
2) По методологии https://zvrba.net/articles/solib-memory-savings.html

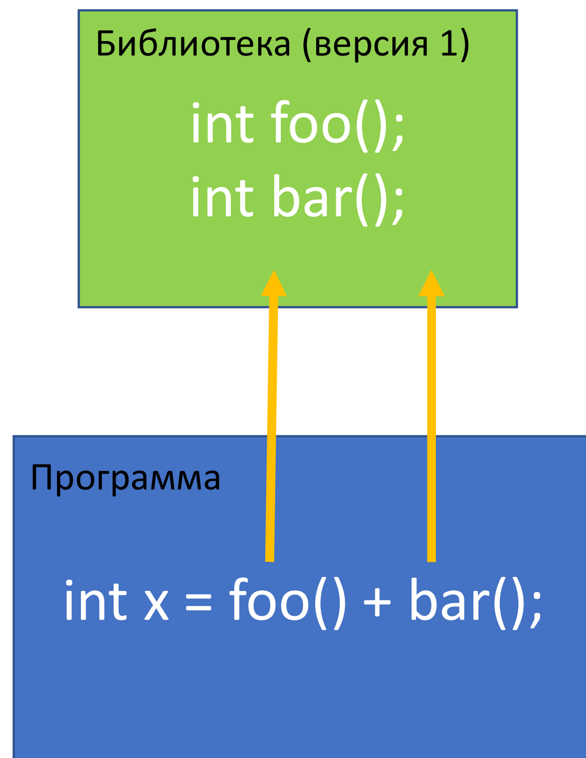
Недостатки DLL

- Накладные расходы
  - Загрузка библиотек
  - Вызовы библиотечных функций
- Более хрупкая инфраструктура (DLL hell)

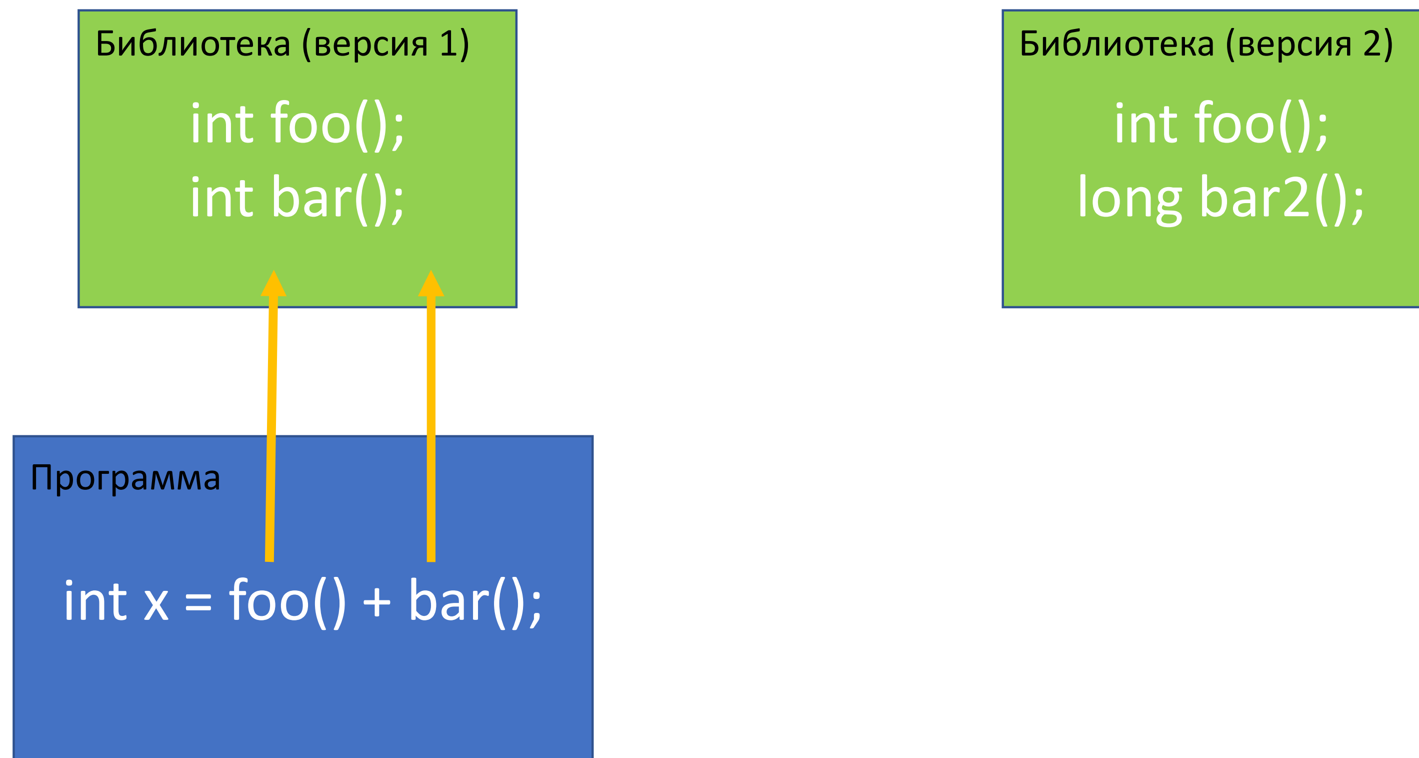
DLL Hell: пример



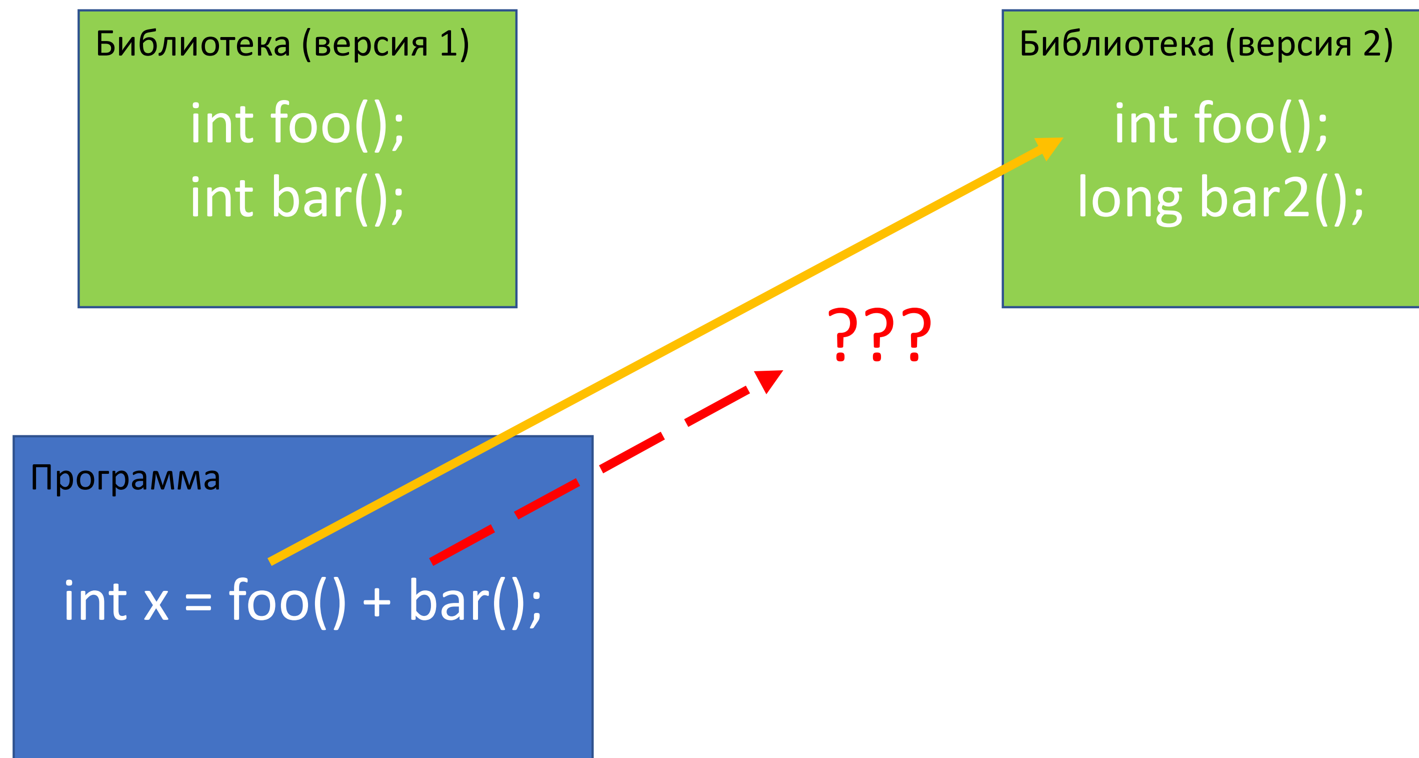
DLL Hell: пример



DLL Hell: пример



DLL Hell: пример



DLL Hell

- При разработке динамических библиотек легко внести *несовместимые изменения*
  - Удаление функции, изменение сигнатуры функции
- Приложения, использовавшие старую версию библиотеки, не смогут работать с новой
  - Не загрузятся или упадут в процессе работы

DLL Hell: решение

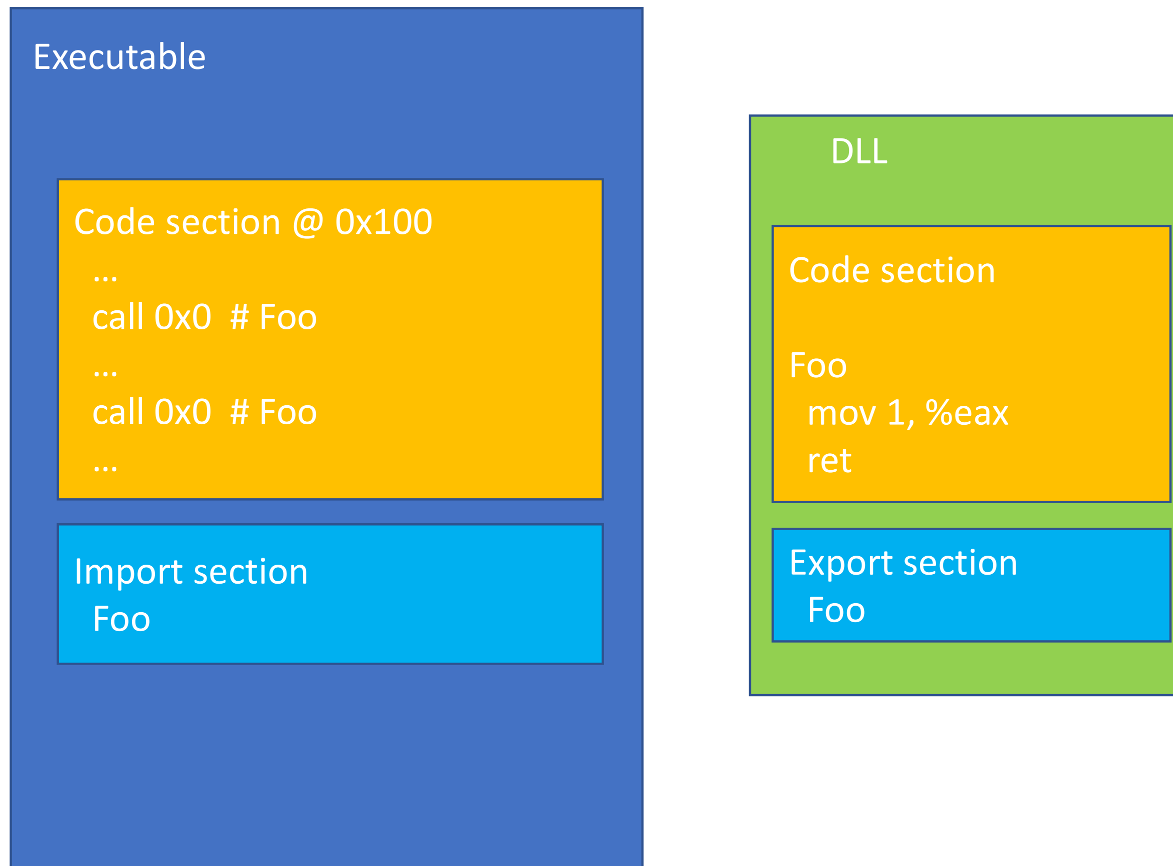
- Разработчик библиотеки должен избегать несовместимых изменений
  - Поиск таких изменений можно автоматизировать (libabigail, ABI Compliance Checker, etc.)
- Если они необходимы разработчик должен обновить в файле библиотеки информацию о её версии
  - SONAME в Linux, DLL manifests в Windows
- Это позволит ОС определить какая версия библиотеки нужна программе и загрузить именно её
- Детали зависят от операционной системы

Принципы работы динамических библиотек

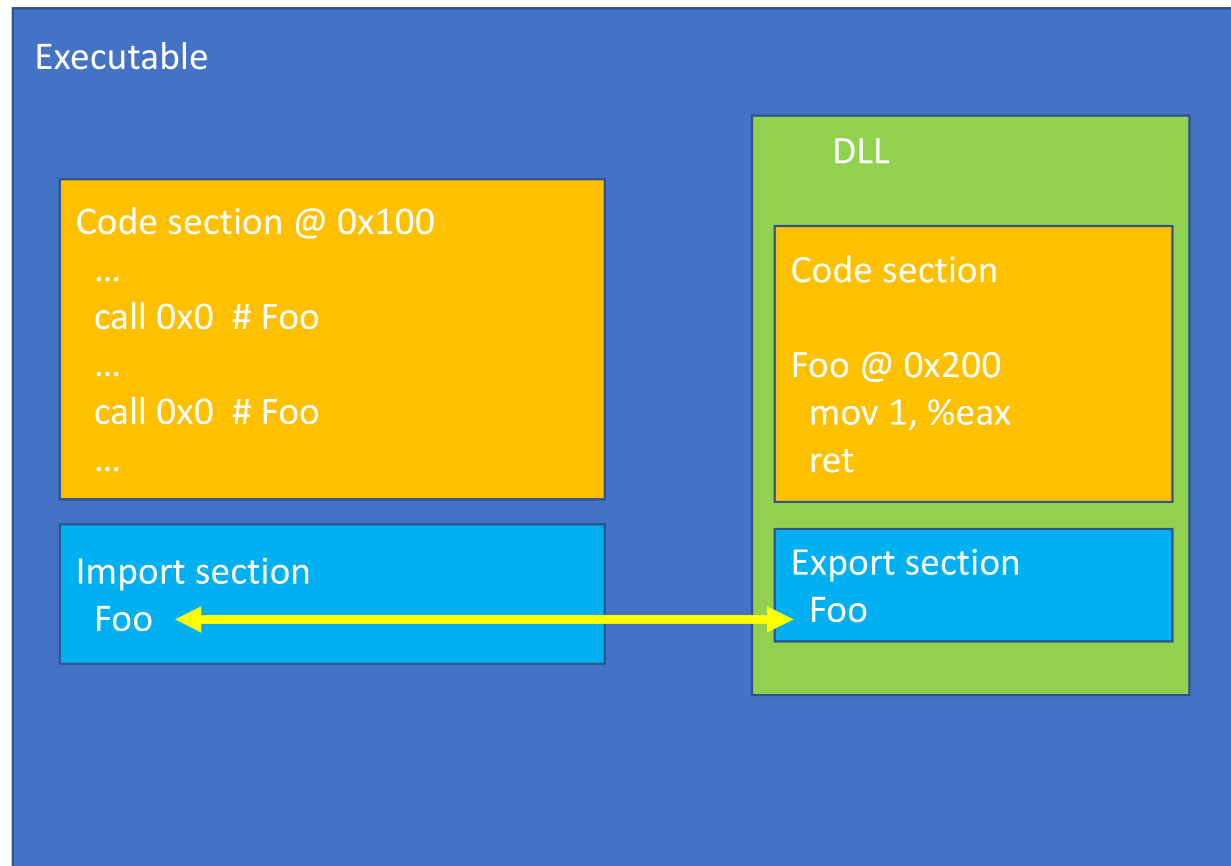
Общие принципы работы DLL

- DLL имеет тот же формат что и исполняемый файл
  - Portable Executable на Windows, ELF на Linux
- Библиотека хранит экспортируемые символы в специальной таблице экспорта
  - .edata на Windows, .dynsym на Linux
- Исполняемый файл хранит список библиотек и импортируемых из них символов в своей таблице импорта
  - .idata на Windows, .dynsym/.dynamic на Linux

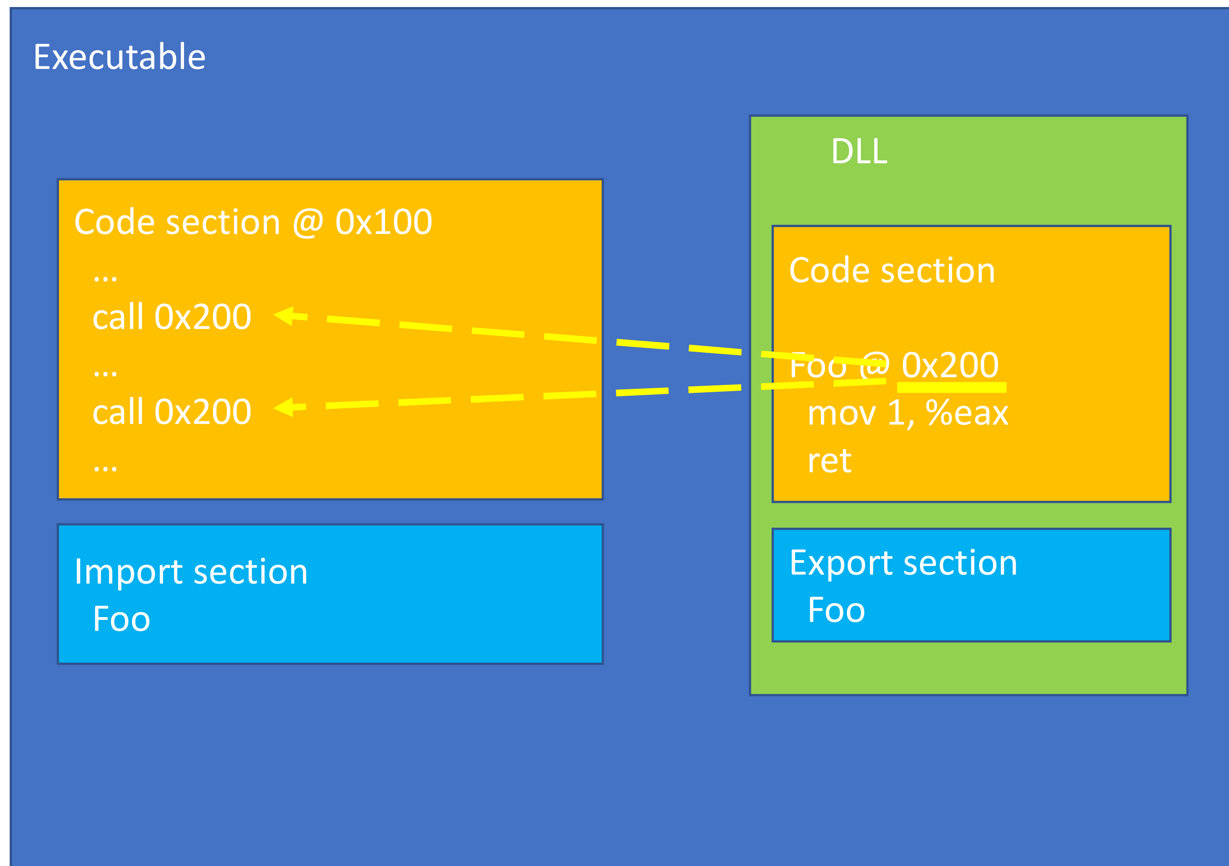
Общие принципы работы DLL



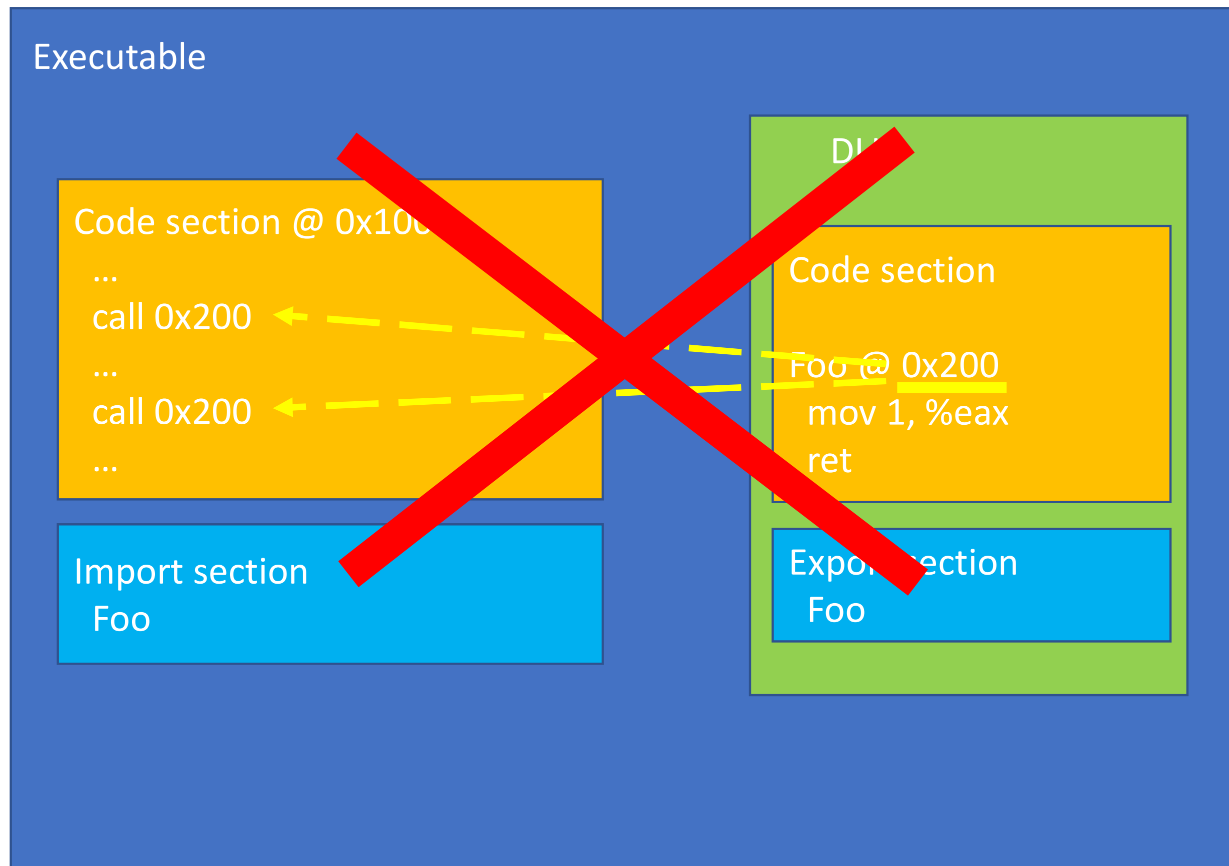
Общие принципы работы DLL



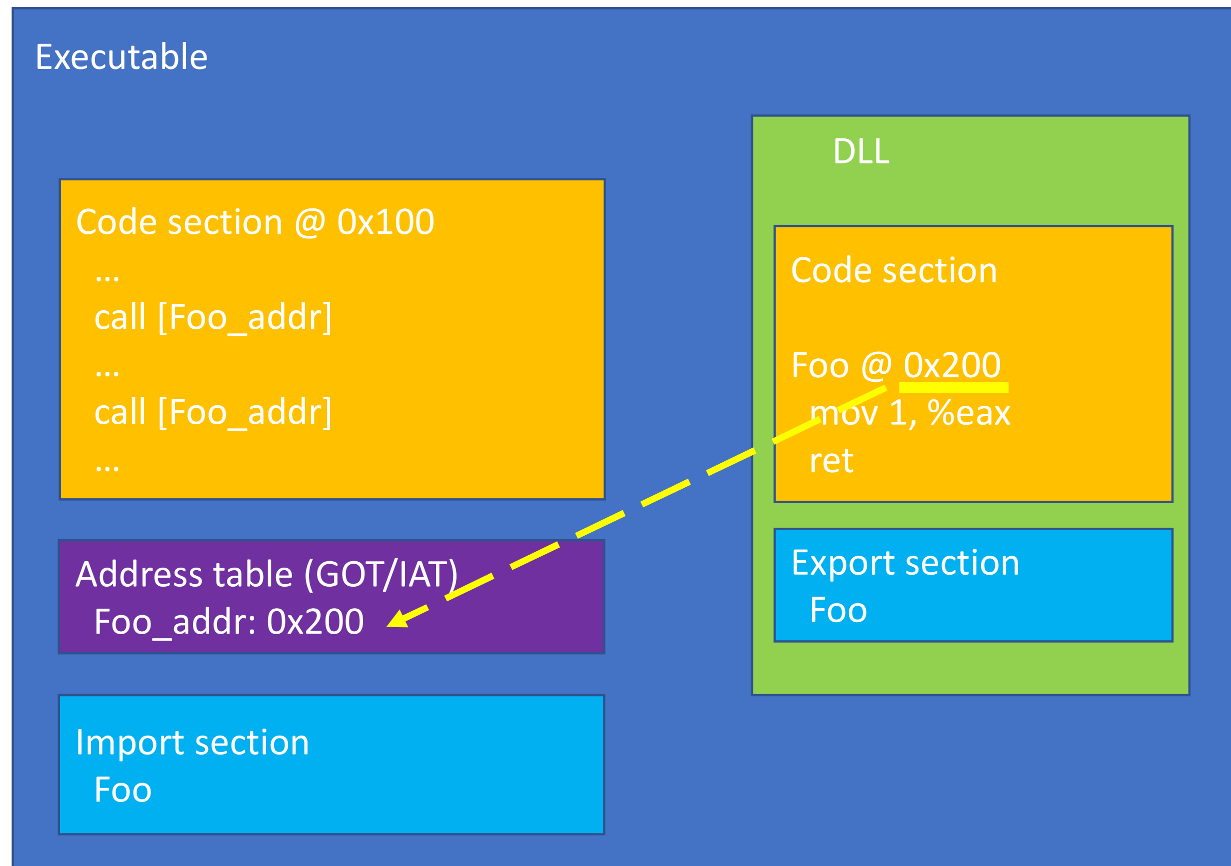
Общие принципы работы DLL



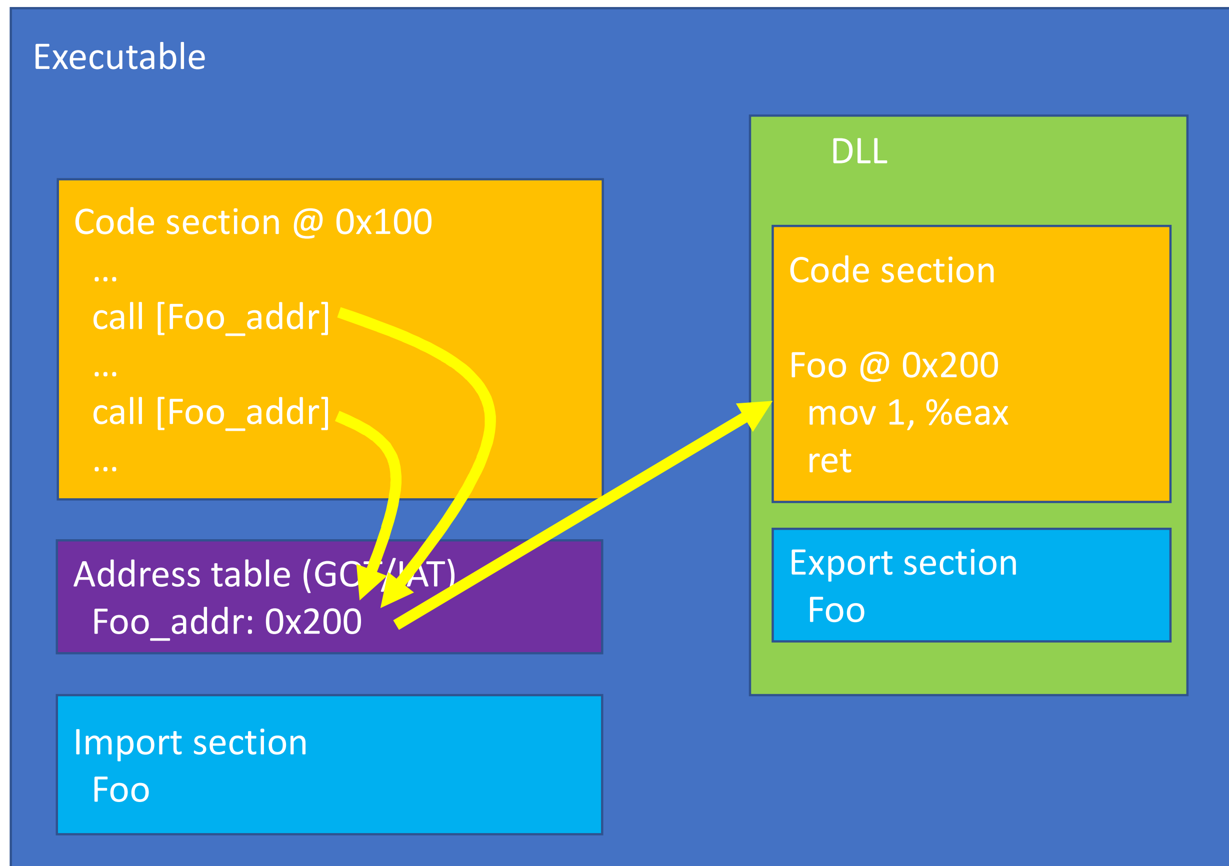
Общие принципы работы DLL



Общие принципы работы DLL



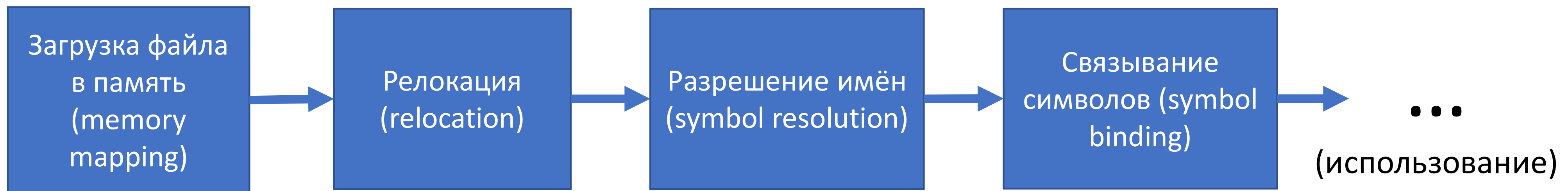
Общие принципы работы DLL



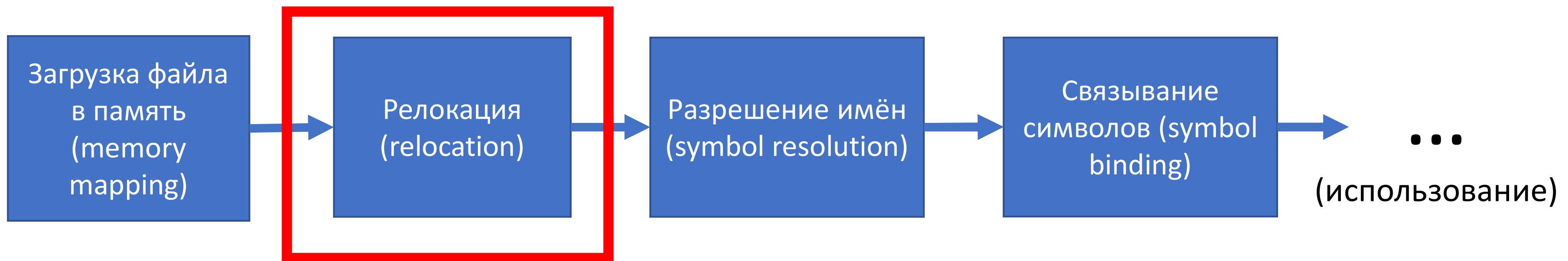
Динамический загрузчик

- Загрузку библиотек осуществляет динамический загрузчик (dynamic loader)
  - /lib64/ld-linux-x86-64.so.2 на Linux
  - Image loader (Ldr) на Windows
- При запуске приложения ядро ОС размещает загрузчик в памяти процесса и передаёт ему управление
- Загрузчик
  - Размещает в памяти файлы импортируемых библиотек
  - Находит (resolves) и связывает (binds) экспортируемые и импортируемые символы
  - Передаёт управление программе

Процесс загрузки DLL



Процесс загрузки DLL



Релокация библиотек: пример

```
$ cat lib.c
int x = 0x12;
int *p = &x;

$ gcc -shared -fPIC lib.c
```

Релокация библиотек: пример

```
$ readelf --dyn-syms a.out
     ...
     5: 0000000000004028     8 OBJECT  GLOBAL DEFAULT   17 p
     6: 0000000000004020     4 OBJECT  GLOBAL DEFAULT   17 x
```

Релокация библиотек: пример

```
$ readelf --dyn-syms a.out
     ...
     5: 0000000000004028      8 OBJECT  GLOBAL DEFAULT   17 p
     6: 0000000000004020      4 OBJECT  GLOBAL DEFAULT   17 x

$ objdump -s -j .data a.out
 4018 18400000 00000000 12000000 00000000  .@..............
 4028 00000000 00000000                    ........
```

X

P

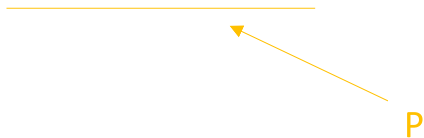
Релокация библиотек

- Адреса глобальных переменных и функций могут быть определены только в рантайме
  - Когда известен точный адрес загрузки библиотеки
- В DLL хранится специальная таблица с адресами указателей, которые должны быть пропатчены после загрузки
  - .rela.dyn в Linux, .reloc в Windows
- Такой процесс патчинга называется *релокацией*

Релокация библиотек: пример

```
$ readelf -r a.out

Relocation section '.rela.dyn' at offset 0x358 contains 8 entries:
...
000000004028  000600000001 R_X86_64_64       0000000000004020 x + 0
```



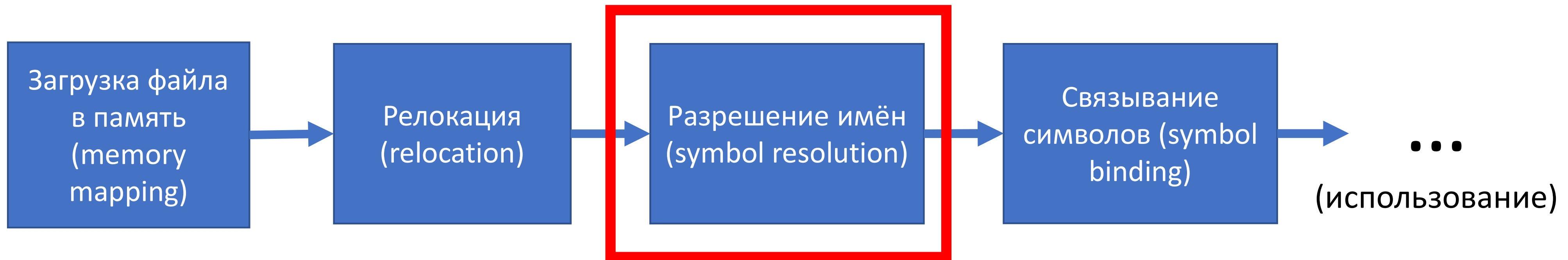
Релокация библиотек: позиционно-независимый код

- Динамические библиотеки компилируются в позиционно-независимый (PC/IP-relative) код
  - В коде отсутствуют абсолютные адреса функций и глобальных переменных
  - Адреса переменных указываются как смещение относительно Program Counter:

```
mov global_var, %rdi    →    mov global_var(%rip), %rdi
```

- Такой код не нужно релоцировать при загрузке
  - Более быстрая загрузка
  - Сегмент кода может разделяться несколькими программами
- Данные по-прежнему нужно релоцировать (например таблицы виртуальных функций)
  - Таких релокаций гораздо меньше

Процесс загрузки DLL



Разрешение имён (symbol resolution)

- Поиск соответствия между экспортируемыми и импортируемыми символами
- Для ускорения поиска информация о символах хранится в хэштаблицах
- Windows и Linux используют разные подходы:
  - Windows: на этапе линковки символ связывается с конкретной библиотекой и может быть загружен только из неё
  - Linux: символ ищется во всех загруженных библиотеках
    - Это делает возможным динамический перехват символов (runtime interposition)

Перехват символов в Linux (runtime interposition)

- Можно заставить загрузчик найти символ не в исходной библиотеке, а в библиотеке-перехватчике
- Обычно перехват символов осуществляется с помощью переменной окружения LD_PRELOAD:

```
$ cat prog.c
int main(int argc) { printf("%d\n", argc); }
$ ./prog a b c
4
$ cat lib.c
int printf(char *fmt, ...) { puts("Hello from interceptor\n"); }
$ LD_PRELOAD=./lib.so ./prog a b c
Hello from interceptor
```

- Частно используется в отладочных инструментах типа Electric Fence или AddressSanitizer для перехвата операций с памятью (malloc и пр.)

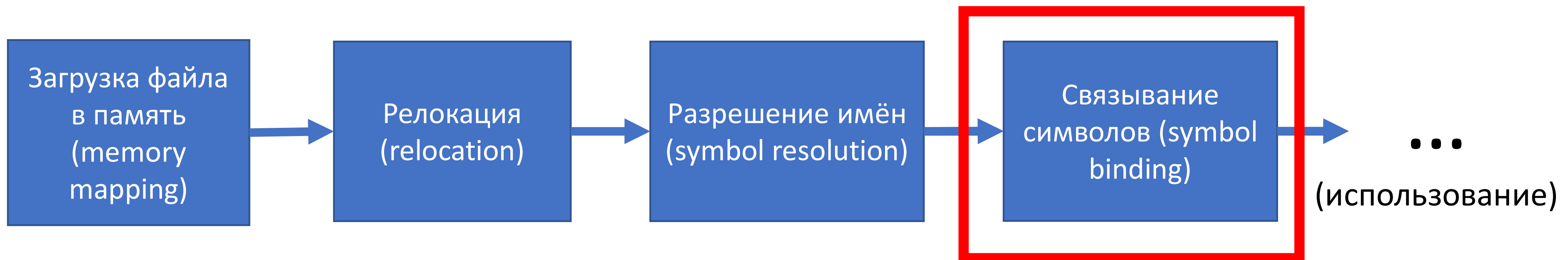
Влияние перехвата на оптимизации

- Компилятору приходится ограничивать оптимизации из-за потенциального перехвата функций
- Компилятор не встраивает вызов функции из-за возможности перехвата foo

```
$ cat mylib.c
void foo() {}
void bar() { foo(); }

$ gcc mylib.c -O3 -fPIC -S -o -
...
bar:
  jmp     foo@PLT
```

Процесс загрузки DLL



Связывание символов (symbol binding)

- Механизм связывания (binding) вызовов функций в исполняемом файле с адресами импортируемых функций, найденными в процессе разрешения имён (symbol resolution)
- Импортируемые функции вызываются через специальную таблицу
  - Import Address Table на Windows, Global Offset Table на Linux
  - Инициализируется загрузчиком *обычно* на старте программы
- Вызов импортируемой функции осуществляется через загрузку адреса из этой таблицы:

```
# Windows
call qword ptr [__imp_foo]

# Linux
call *foo@GOTPCREL(%rip)
```

- Вызов функции из библиотеки является косвенным (indirect)

Ленивое связывание в Linux (lazy binding)

- Загрузка символа из таблицы адресов осуществляется не напрямую, а через функцию-заглушку (PLT stub)
- PLT stubs создаются линкером автоматически
- Откладывает поиск символа до первого его использования

```
    .section .text
    ...
    call foo
    ...
    .section .plt
  foo:    # PLT stub pseudocode
    if (first call)
      GOT[foo] = resolve address of foo
    call GOT[foo]
```

Ускорение работы динамических библиотек

Накладные расходы при использовании DLL

- Загрузка библиотеки
  - Релокация
  - Разрешение и связывание символов
- Работа с библиотекой
  - Косвенные вызовы функций (indirect calls)

Накладные расходы при использовании DLL

- **Загрузка библиотеки**
  - Релокация
  - Разрешение и связывание символов
- Работа с библиотекой
  - Косвенные вызовы функций (indirect calls)

Ускорение загрузки DLL: отключение неиспользуемых библиотек

- Часто в больших программах можно случайно указать лишние библиотеки при сборке
- Их загрузка замедлит работу приложения даже если они не будут использоваться
- Флаг -Wl,--as-needed позволит линкеру проигнорировать такие библиотеки
- Флаг включен по умолчанию в некоторых дистрибутивах (Ubuntu, но не Fedora/RHEL)

Ускорение загрузки DLL: отложенная загрузка библиотек

- Часто библиотека используется только в редких случаях
- Вместо загрузки на старте было бы выгодно загружать её при первом использовании (отложенная загрузка, lazy loading)
- Некоторые платформы предоставляют такую возможность:
  - Windows: флаг /DELAYLOAD
  - macOS: флаг -Wl,-z,-lazy-l (больше не поддерживается)
- Для Linux стандартного решения нет, но можно использовать утилиту Implib.so
  - https://github.com/yugr/Implib.so

Implib.so

- Реализует отложенную загрузку в POSIX-систем
- Для заданной DLL генерирует небольшую статическую библиотеку с функциями-заглушками (trampolines)
- Вместо DLL программа линкуется с этой статической библиотекой
- Во время работы вызов функции-заглушки приведёт к загрузке библиотеки и передаче управления в неё:

```
int foo(type1 arg1, type2 arg2, ...) {  # Stub
  static void *foo_real = NULL;
  if (!foo_real) {
    void *handle = dlopen(...);
    foo_real = dlsym(handle, "foo");
  }
  return foo_real(arg1, arg2, ...);
}
```

Implib.so

- Реализована с помощью API динамической загрузки POSIX (dlopen, dlsym)
- Имеет минимальные накладные расходы
- Поддерживает большое количество платформ
  - x86, ARM, AArch64, RISC-V, e2k, etc.
  - Linux (+ частично BSD)

Накладные расходы при использовании DLL

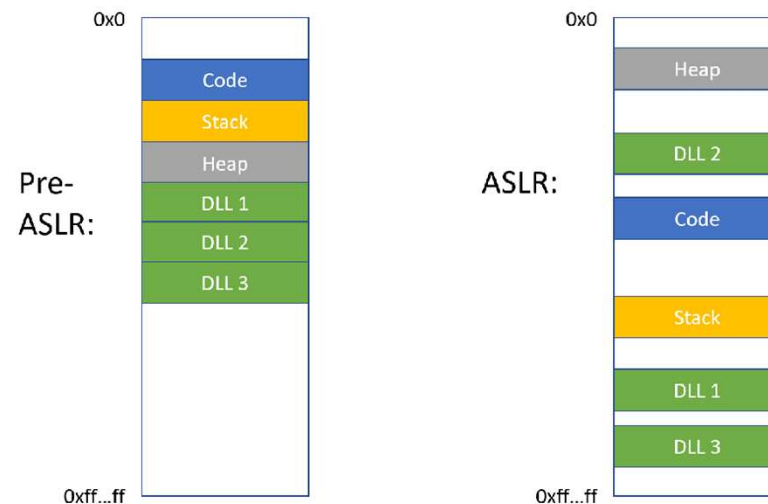
- Загрузка библиотеки
  - **Релокация**
  - Разрешение и связывание символов
- Работа с библиотекой:
  - Косвенные вызовы функций (indirect calls)

Ускорение загрузки DLL: link-time relocation

- Релокации можно избежать если выбрать адрес загрузки на этапе линковки:
  - Просканировать все установленные программы и библиотеки
  - Статически распределить адресное пространство между всеми библиотеками
  - Слинковать каждую библиотеку по выделенному ей адресу
- Динамический загрузчик сможет избежать релокации библиотеки
- Решение:
  - Windows: preferred load address (параметр /BASE)
  - Linux: Prelink

Ускорение загрузки DLL: link-time relocation

- Оптимизация нерелевантна из-за современных требований к безопасности:
    - Механизм Address-Space Layout Randomization требует загружать DLL по случайным адресам (для усложнения подбора адресов хакерами)



Релокация библиотек: оптимизация в Windows

- Для ускорения работы библиотека во всех процессах загружается по одному и тому же адресу
- Накладные расходы возникают только при первой загрузке
- Работает только в современных версиях Windows

Накладные расходы при использовании DLL

- Загрузка библиотеки
  - Релокация
  - **Разрешение и связывание символов**
- Работа с библиотекой:
  - Косвенные вызовы функций (indirect calls)

Ускорение работы DLL: prelinking

- Заранее (до запуска) инициализировать таблицу адресов в файле программы
- Ускорит поиск символов если библиотека всегда загружается по одному и тому же адресу
  - Т.е. была проведена link-time relocation
- Решение:
  - Windows: DLL binding
  - Linux: Prelink
- Не используется в современных версиях Windows и Linux из-за ASLR

Ускорение работы DLL: оптимизация таблиц символов

- Поиск символов в Linux осуществляется по хэштаблицам, хранящимся в файлах динамических библиотек
- Линкеры позволяют управлять размером и форматом этих хэштаблиц
- Обычно рекомендуемая конфигурация опций:
  - -Wl,--hash-style=both -Wl,-O1
- -Wl,--hash-style=both уже включена по умолчанию во всех современных дистрибутивах
- -Wl,-O1 не оказывает существенного влияния на производительность

Ускорение работы DLL: отключение ленивого связывания

- Ленивое связывание в Linux ускоряет загрузку библиотек ценой накладных расходов в процессе работы
- К загрузке адреса и косвенному вызову функции добавляется вызов PLT-заглушки
  - Лишний jump
  - Вырастает нагрузка на кэши и branch predictor
  - Загрузку адреса приходится осуществлять при каждом вызове
- Ленивая загрузка и связанные с ней накладные расходы могут быть отключены флагом -fno-plt

Ускорение работы DLL: отключение ленивого связывания

- Использование -fno-plt
  - Ускоряет вызовы библиотечных функций
  - Снижает нагрузку на I$ и BTB
  - Замедляет загрузку библиотеки (т.к. все адреса надо инициализировать на старте программы)
- Современные требования к безопасности и так рекомендуют разрешать все функции на старте программы
  - Позволяет использовать технологию Full Relro (-Wl,-z,relro) для защиты от непреднамеренных модификаций GOT
  - Full Relro используется по умолчанию в RHEL/Fedora и Ubuntu

Ускорение работы DLL: отключение ленивого связывания

- Примеры:
  - Использование -fno-plt в Clang даёт до 10% прироста производительности

Накладные расходы при использовании DLL

- Загрузка библиотеки
  - Релокация
  - Разрешение и связывание символов
- Работа с библиотекой
  - **Косвенные вызовы функций (indirect calls)**

Проблема с экспортируемыми символами в Linux

- По умолчанию на Linux все функции в DLL экспортируются
  - Для совместимости со статическими библиотеками
- Вызовы даже внутренних функций библиотеки происходят через таблицу адресов (GOT)
  - Из-за возможности перехвата символов
- Накладные расходы:
  - Косвенные вызовы функций
  - Отмена оптимизаций в компиляторе (inlining, cloning, etc.)

Ускорение работы DLL: отключение перехвата функций

- Флаги компилятора позволяют отключить учёт перехвата
- -Bsymbolic/-Bsymbolic-functions – заменяет внутренние вызовы экспортируемых функций на прямые на этапе линковки
  - Опция включена по умолчанию в некоторых дистрибутивах (Ubuntu, но не Debian)
- -fno-semantic-interposition – игнорирует возможность перехвата на этапе компиляции
  - Включена по умолчанию в Clang, но не в GCC
  - Включается в GCC под -Ofast
- Для оптимальной производительности требуются оба флага

Ускорение работы DLL: отключение перехвата функций

- Примеры:
  - Использование -Bsymbolic-functions при сборке Clang даёт до 10% прироста производительности
  - Использование -fno-semantic-interposition при сборке Python даёт до 30% прироста производительности
    - https://fedoraproject.org/wiki/Changes/PythonNoSemanticInterpositionSpeedup

Ускорение работы DLL: сокращение интерфейса библиотеки

- Простой способ ускорения работы
- Явно помечаем публичные функции библиотеки:

```
$ cat mylib.c
void internal() {}

__attribute__((visibility("default")))
void public() { internal(); }

$ gcc mylib.c -fvisibility=hidden -fPIC -shared
```

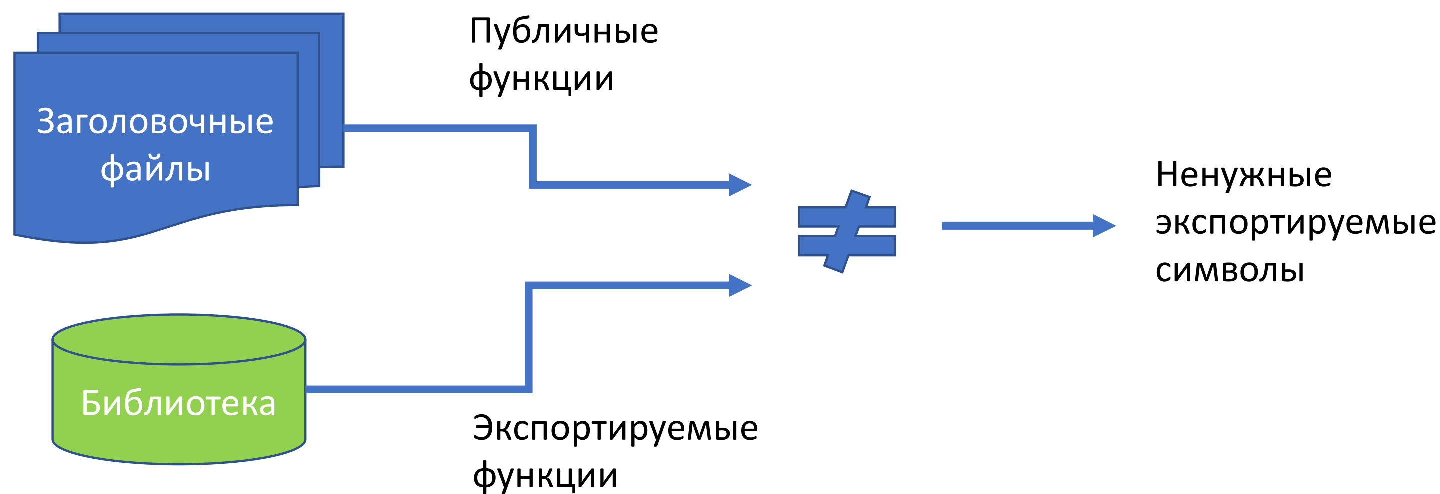
- Непубличные функции будет полноценно оптимизироваться компилятором
- Какие функции экспортировать?
  - Как правило это функции из публичных заголовочных файлов
  - Таких функций очень немного по сравнению со всеми функциями библиотеки

Сокращение интерфейса библиотек в дистрибутивах

- При наличии большой кодовой базы (например дистрибутива) может быть трудно найти библиотеки с избыточными экспортами
- Поиск таких библиотек можно автоматизировать с помощью утилиты ShlibVisibilityChecker
  - https://github.com/yugr/ShlibVisibilityChecker

ShlibVisibilityChecker

- Анализирует функции в публичных заголовочных файлах библиотеки с помощью libclang
- Сравнивает их с функциями, экспортируемыми библиотекой
- Избыточные экспорты должны быть скрыты



Пример использования ShlibVisibilityChecker

```
$ read_header_api --only-args /usr/include/x86_64-linux-gnu/gmp.h > api.txt

$ read_binary_api --permissive /usr/lib/x86_64-linux-gnu/libgmp.so.10.4.1 > abi.txt

$ diff api.txt abi.txt | wc -l
323

$ diff api.txt abi.txt
0a1,10
> __gmp_0
> __gmp_allocate_func
> __gmp_asprintf_final
> __gmp_asprintf_funs
…
```

Резюме

Резюме

- Динамические библиотеки имеют ряд преимуществ над статическими

Резюме

- Динамические библиотеки имеют ряд преимуществ над статическими
- Добавляют накладные расходы при загрузке и во время работы приложения

Резюме

- Динамические библиотеки имеют ряд преимуществ над статическими
- Добавляют накладные расходы при загрузке и во время работы приложения
- Современные тулчейны содержат средства, позволяющие существенно снизить оверхед
    - Особенно на Linux

Что почитать?

- Linkers, Loaders and Shared Libraries in Windows, Linux, and C++ (Ofek Shilon, CppCon 2023)
  - https://www.youtube.com/watch?v=_enXuIxuNV4
  - Общий обзор DLL на разных платформах

- How to Write Shared Libraries (by Ulrich Drepper)
  - https://www.akkadia.org/drepper/dsohowto.pdf
  - Всё что нужно знать о DLL на Linux

- Everything You Ever Wanted to Know about DLLs (by James McNellis, CppCon 2017)
  - https://www.youtube.com/watch?v=JPQWQfDhICA
  - Всё что нужно знать о DLL на Windows

- MaskRay Blog
  - https://maskray.me/blog
  - Блог о системном программировании под Linux (GOT, PLT, etc.)

Спасибо за внимание!

Проверка экономии памяти

- Собрать сканнер
  - gcc -Wall -Wextra scripts/ram-savings.c
- Запустить под sudo:
  - sudo ./a.out

Анализ экономии диска

- Запустить
  - scripts/disk-savings.pl
- Скрипт даёт верхнюю оценку – реальная экономии памяти будет ниже
  - При использовании статическх библиотек не все их функции будут использованы приложениями
  - Соответственно библиотеки только частично будут включены в исполняемые файлы

Проверка -Wl,-O1

- Собрать две версии LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_SHARED_LINKER_FLAGS='-Wl,-O1’
- Сравнить производительность:
  - ./benchmark.pl 10 path/to/clang -h

Проверка -fno-plt

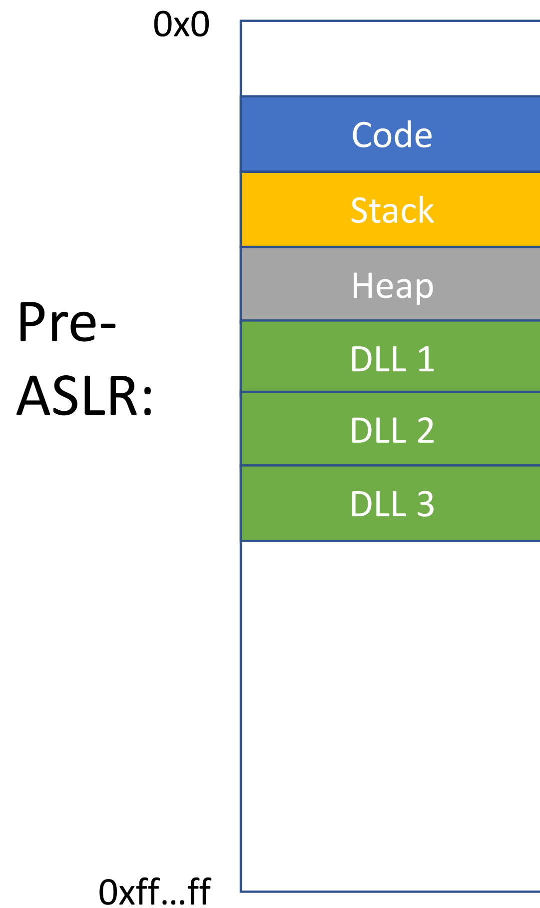
- Собрать две версии LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_CXX_FLAGS='-fno-plt'
- Сравнить производительность:
  - ./benchmark.pl 10 path/to/clang -S -O2 ~/InstCombining.ii

Проверка -Bsymbolic-functions

- Собрать две версии LLVM:
  - -DBUILD_SHARED_LIBS=ON
  - -DBUILD_SHARED_LIBS=ON -DCMAKE_SHARED_LINKER_FLAGS='-Wl,-Bsymbolic-functions’
- Сравнить производительность:
  - ./benchmark.pl 10 path/to/clang -S -O2 ~/InstCombining.ii

Address-space Layout Randomization

Pre-ASLR:



ASLR: