План

- Динамические библиотеки и динамическая компоновка
- Перехват вызовов функций
- Позиционно независимый код

Статическая компоновка

- Зачем нужно
  - Модульность
  - Эффективность
- Что происходит
  - Разрешение символов (resolving)
  - Перемещение (relocation)
- Форматы файлов для хранение объектного кода
  - Executable and Linkable Format (ELF)
  - Microsoft Portable Executable and Common Object File Format (PE/COFF)
- Как работает
  - ELF – секции `.symtab`, `.rel.text`, `.rel.data`
  - PE/COFF – секции `.reloc`, `.edata`, `.idata`
- Набор логически связанных объектных файлов оформляется в виде статической библиотеки

Динамические (разделяемые) библиотеки

- Статические библиотеки имеют следующие недостатки:
  - Многократное копирование кода в построенных исполняемых файлах (всем нужно std libc)
  - Копии кода в исполняющихся программах
  - Любое исправление в системных библиотеках требует повторной компоновки для **всех** приложений

- Способ преодолеть эти недостатки: динамические библиотеки (shared libraries)
  - Объектные файлы, в которых содержатся код и данные компонуются с приложением *динамически*, либо во время *загрузки*, либо во время *выполнения*
  - Практикуются названия: DLL-ки, `.so`-шники

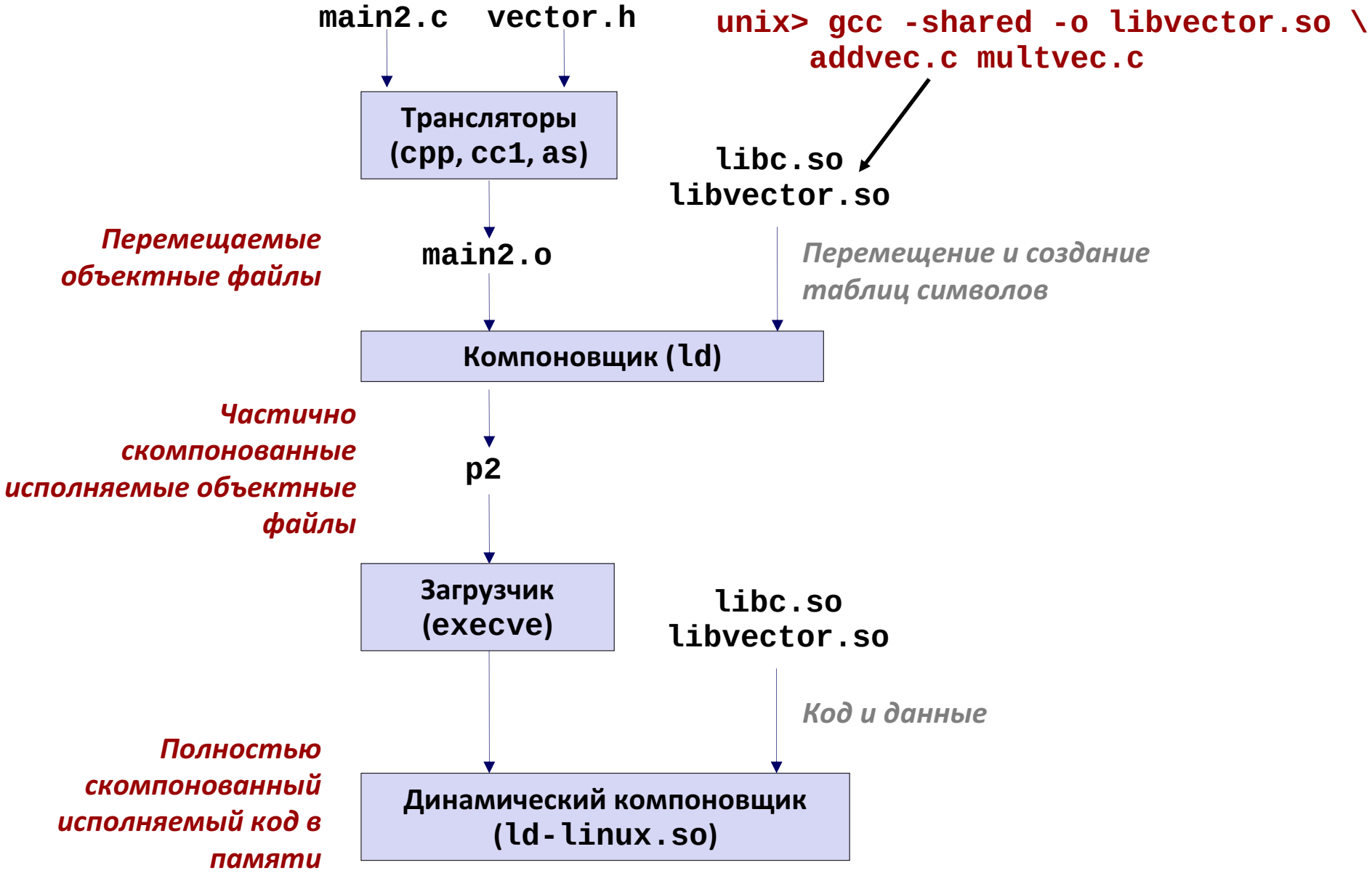
Динамические (разделяемые) библиотеки

- Динамическая компоновка происходит когда исполняемый файл в первый раз загружается и начинает работать (компоновка во время загрузки).
  - В Linux наиболее распространено, автоматически выполняется динамическим компоновщиком (**`ld-linux.so`**).
  - Стандартная библиотека языка Си (**`libc.so`**) обычно компонуется динамически.
- Динамическая компоновка может происходить когда программа уже работает (компоновка времени выполнения).

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
```

- Функции динамических библиотек могут одновременно использоваться несколькими процессами.

Динамическая компоновка времени загрузки



Динамическая компоновка времени исполнения

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main() {
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /*
     * Динамически загружаем библиотеку,
     * содержащую функцию addvec()
     */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
                fprintf(stderr, "%s\n", dlerror());
                exit(1);
    }
    ...
}
```

Динамическая компоновка времени исполнения

```
    ...

    /* получить указатель на функцию addvec() */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
                fprintf(stderr, "%s\n", error);
                exit(1);
    }

    /* Теперь можно вызывать addvec() как обычную функцию */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Выгружаем динамическую библиотеку из памяти */
    if (dlclose(handle) < 0) {
                fprintf(stderr, "%s\n", dlerror());
                exit(1);
    }
    return 0;
}
```

Пример: перехват библиотечных функций

- Library interpositioning : прием, использующий стандартные механизмы компоновки, для перехвата вызова произвольных функций
- Способы перехвата вызовов:
  - Во время компиляции
  - Во время компоновки: когда перемещаемый объектный файл компонуется в исполняемый объектный файл
  - Загрузка/выполнение: когда исполняемый объектный файл загружается в память, динамически компонуется, выполняется.
- Инструментирование кода

Практическое использование перехвата вызовов

- Информационная безопасность
  - Ограничение возможностей («песочница»)
    - Перехват вызовов функций из libc.
  - Скрытое шифрование
    - Автоматическое шифрование сетевого траффика.
- Мониторинг и профилирование
  - Подсчет количество вызовов функций
  - Анализ мест вызова функций и фактических аргументов
  - Трассировка функций работы с динамической памятью: malloc и т.п.
    - Обнаружение утечек памяти
    - **Построение трассы с адресами памяти**

Пример

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main()
{
    free(malloc(10));
    printf("hello, world\n");
    exit(0);
}
```

hello.c

- Цель: построить трассу с адресами и размерами выделяемых и освобождаемых блоков без модификации исходного кода.

- Три возможных решения: перехватывать функции `malloc` и `free` во время компиляции, компоновки, загрузки/выполнения.

Перехват во время компиляции

```
#ifdef COMPILETIME
/*
 * Перехват malloc и free во время компиляции с
 * использованием препроцессора.
 */

#include <stdio.h>
#include <malloc.h>

/*
 * mymalloc – обертка для malloc
 */
void *mymalloc(size_t size, char *file, int line)
{
    void *ptr = malloc(size);
    printf("%s:%d: malloc(%d)=%p\n", file, line, (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Перехват во время компиляции

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__ )
#define free(ptr) myfree(ptr, __FILE__, __LINE__ )

void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);
```

malloc.h

```
linux> make helloc
gcc -O2 -Wall -DCOMPILETIME -c mymalloc.c
gcc -O2 -Wall -I. -o helloc hello.c mymalloc.o
linux> make runc
./helloc
hello.c:7: malloc(10)=0x501010
hello.c:7: free(0x501010)
hello, world
```

Перехват во время компоновки

```
#ifdef LINKTIME
/*
 * Перехват во время компоновки с использованием
 * статического компоновщика (ld) и флага "--wrap symbol"
 */

#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/*
 * __wrap_malloc – обертка для malloc
 */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Перехват во время компоновки

```
linux> make hellol
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hellol hello.c mymalloc.o
linux> make runl
./hellol
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- Флаг “`-Wl`” передает аргумент компоновщику
- Ключ “`--wrap,malloc`” требует разрешать ссылки следующим способом:
  - Ссылка `malloc` должна быть разрешена как `__wrap_malloc`
  - Ссылка `__real_malloc` должна быть разрешена как `malloc`

Перехват времени выполнения

```
#ifdef RUNTIME
 /* Перехват времени выполнения функций malloc и free, использующий
  * переменную LD_PRELOAD и динамический компоновщик (ld-linux.so) */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size)
{
    static void *(*mallocp)(size_t size);
    char *error;
    void *ptr;

    /* получить адрес malloc из libc */
    if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(1);
        }
    }
    ptr = mallocp(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Перехват времени выполнения

```
linux> make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c
gcc -O2 -Wall -o hellor hello.c
linux> make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- Переменная окружения LD_PRELOAD требует от динамического компоновщика искать неразрешенные ссылки в первую очередь в (т.к., `malloc`) в библиотеках `libdl.so` и `mymalloc.so`.
  - **`libdl.so`** необходима для разрешения ссылки на функцию **`dlopen`**.

Перехват: Итоги

- Во время компиляции
  - Вызовы функций malloc/free разворачиваются препроцесосром в вызовы mymalloc/myfree
- Во время компоновки
  - Используется возможность компоновщика менять определенные символы
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- Во время выполнения
  - Реализуется собственная версия malloc/free, использующая динамическую компоновку.

Разделяемый код

- В каждом процессе повторяются библиотеки
  - libc, libstdc++ и др.
- Один экземпляр в физической памяти – несколько экземпляров в виртуальных адресных пространствах разных процессов
- Специальные правила оформления разделяемых библиотек
  - Linux – Position Independent Code (PIC)
  - Win – memory mapping. Полноценное разделение кода между процессами не гарантируется.

Пример – обычный код

```
gcc -S -O2 -fomit-frame-pointer foo.c
```

```
int foo() {
   static int i = 0;
   return ++i;
}
```

```
 .file   "foo.c"
   .text
   .p2align 4,,15
.globl foo
   .type   foo, @function
foo:
   movl    i.1178, %eax
   addl    $1, %eax
   movl    %eax, i.1178
   ret
   .size   foo, .-foo
   .local  i.1178
   .comm   i.1178,4,4
   .ident  "GCC: (GNU) 4.3.3"
   .section        .note.GNU-stack,"",@progbits
```

Пример – обычный код

```
gcc -S -O2 -fomit-frame-pointer foo.c
```

```
int foo() {
   static int i = 0;
   return ++i;
}
```

```
section .bss
   i.1178 resd 1
section .text
global foo
foo:
   mov    eax, dword [i.1178]
   add    eax, 1
   mov    dword [i.1178], eax
   ret
```

Код переписан с использованием синтаксиса nasm

Пример – получить текущий EIP

```
func:
   call .label
.label:
   pop  eax
   ret
```

Пример – получить текущий EIP

```
func:
   call .label
.label:
   pop  eax
   add  eax, func - .label
   ret
```

Пример – получить текущий EIP

```
#include <stdio.h>

asm (
        ".text\n"
        "fun:\n"
        "call .label\n"
        ".label:\n"
        "popl %eax\n"
        "addl $fun-.label, %eax\n"
        "ret\n"
        ".previous\n"
);

void *fun ();

int main() {
        printf("address of function: %p\n", &fun);
        printf("the function returns: %p\n", fun());
        return 0;
}
```

Пример – переписываем обычный код

```
int foo() {
   static int i = 0;
   return ++i;
}
```

```
# AT&T syntax
.bss
i.1178:
.skip 4
.text
.global foo
foo:
call .label
.label:
popl %ecx                       # ecx содержит адрес .label
addl $i.1178-.label, %ecx       # ecx содержит адрес i.1178
incl (%ecx)
movl (%ecx), %eax
ret
```

Дизассемблируем foo.o

```
objdump -rd -M intel foo.o
```

```
foo.o:     file format elf32-i386

Disassembly of section .text:

00000000 <foo>:
   0:   a1 00 00 00 00          mov    eax,ds:0x0
                        1: R_386_32     .bss
   5:   83 c0 01                add    eax,0x1
   8:   a3 00 00 00 00          mov    ds:0x0,eax
                        9: R_386_32     .bss
   d:   c3                      ret
```

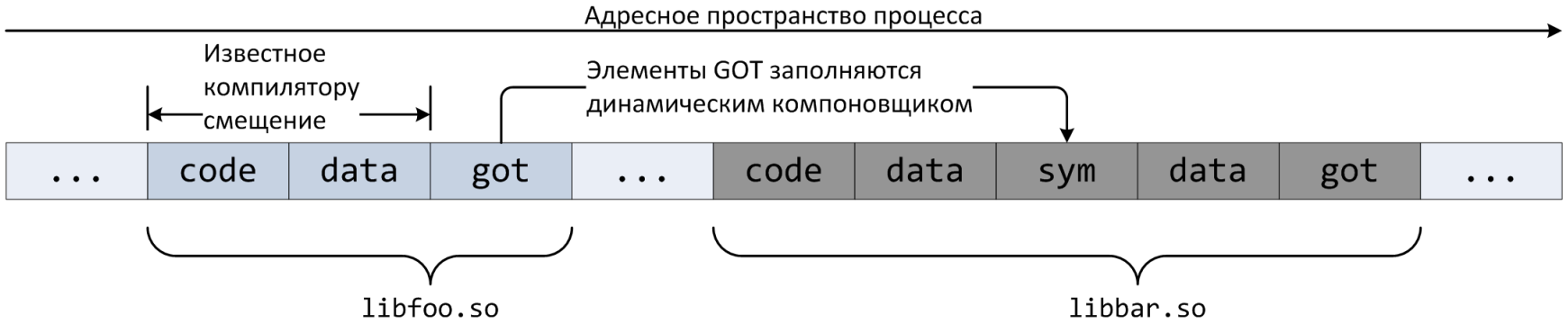
Дизассемблируем foo_pic.o

```
objdump -rd -M intel foo_pic.o
```

```
foo_pic.o:     file format elf32-i386

Disassembly of section .text:

00000000 <foo>:
   0:   e8 00 00 00 00          call   5 <.label>

00000005 <.label>:
   5:   59                      pop    ecx
   6:   81 c1 03 00 00 00       add    ecx,0x3
                        8: R_386_PC32   .bss
   c:   ff 01                   inc    DWORD PTR [ecx]
   e:   8b 01                   mov    eax,DWORD PTR [ecx]
  10:   c3                      ret
```

Global Offset Table

- Необходимо ссылаться на внешние переменные
  - Их адрес становится известен только во время динамической компоновки
- Код библиотеки нельзя модифицировать, секции данных можно
- Размещаем таблицу с адресами внешних символов в известном месте – на фиксированном смещении от начала библиотеки
- Отдельная секция – .got



Пример

```
extern int sym;

int foo() {
  return sym;
}
```
foo-so.c

```
gcc -S –O2 –fomit-frame-pointer
-fPIC -masm=intel foo-so.c
```

```
foo:
        call    __i686.get_pc_thunk.cx
        add     ecx, _GLOBAL_OFFSET_TABLE_
        mov     eax, dword [sym@GOT + ecx]
        mov     eax, dword [eax]
        ret

__i686.get_pc_thunk.cx:
        mov     ecx, dword [esp]
        ret
```
foo-so.s

Код переписан с использованием синтаксиса nasm

Procedure Linkage Table

- GOT должна быть полностью заполнена при динамической компоновке
  - Не все переменные могут реально использоваться
- Обращение к функциям происходит только через инструкцию call – возможно выполнить **ленивое связывание** (lazy binding)
  - В GOT вместо адреса реальной функции помещается адрес функции-заглушки
  - При первом вызове заглушка выполняет поиск адреса реальной функции, помещает его в GOT вместо своего и производит прыжок по этому адресу
  - Все следующие вызовы используют реальный адрес из GOT
- Заглушки размещаются в секции .plt
  - По соглашению при вызове заглушки ebx должен содержать адрес GOT

Пример

```
extern int bar();

int foo() {
  return bar();
}
```
foo-so-func.c

```
gcc -S –O2 –fomit-frame-pointer
-fPIC -masm=intel foo-so-func.c
```

```
foo:
        push    ebx
        call    __i686.get_pc_thunk.bx
        add     ebx, _GLOBAL_OFFSET_TABLE_
        sub     esp, 8
        call    bar@PLT
        add     esp, 8
        pop     ebx
        ret

__i686.get_pc_thunk.bx:
        mov     ebx, dword [esp]
        ret
```

Код переписан с использованием синтаксиса nasm