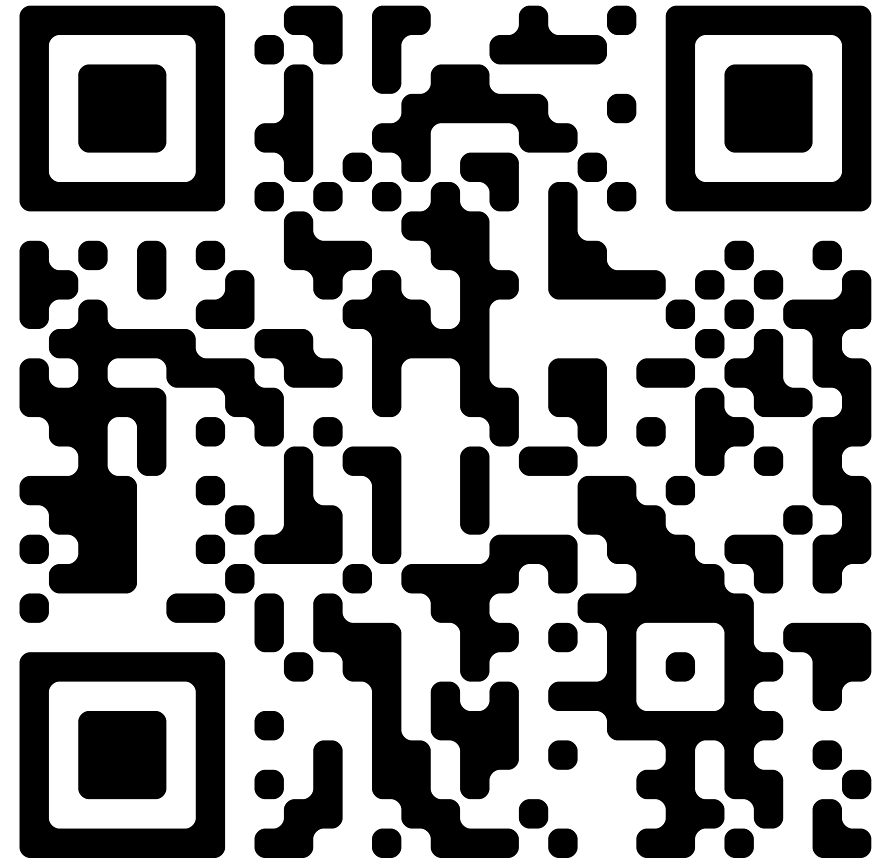
Связывание

Основы информатики.

Компьютерные основы программирования

goo.gl/X7evF



На основе **CMU** 15-213/18-243:

Introduction to Computer Systems

goo.gl/TDDVV

Лекция **11**, **16** апреля, 2017

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

dseverov@mail.mipt.ru

cs.mipt.ru/wp/?page_id=346

Пример Си-программы

main.c

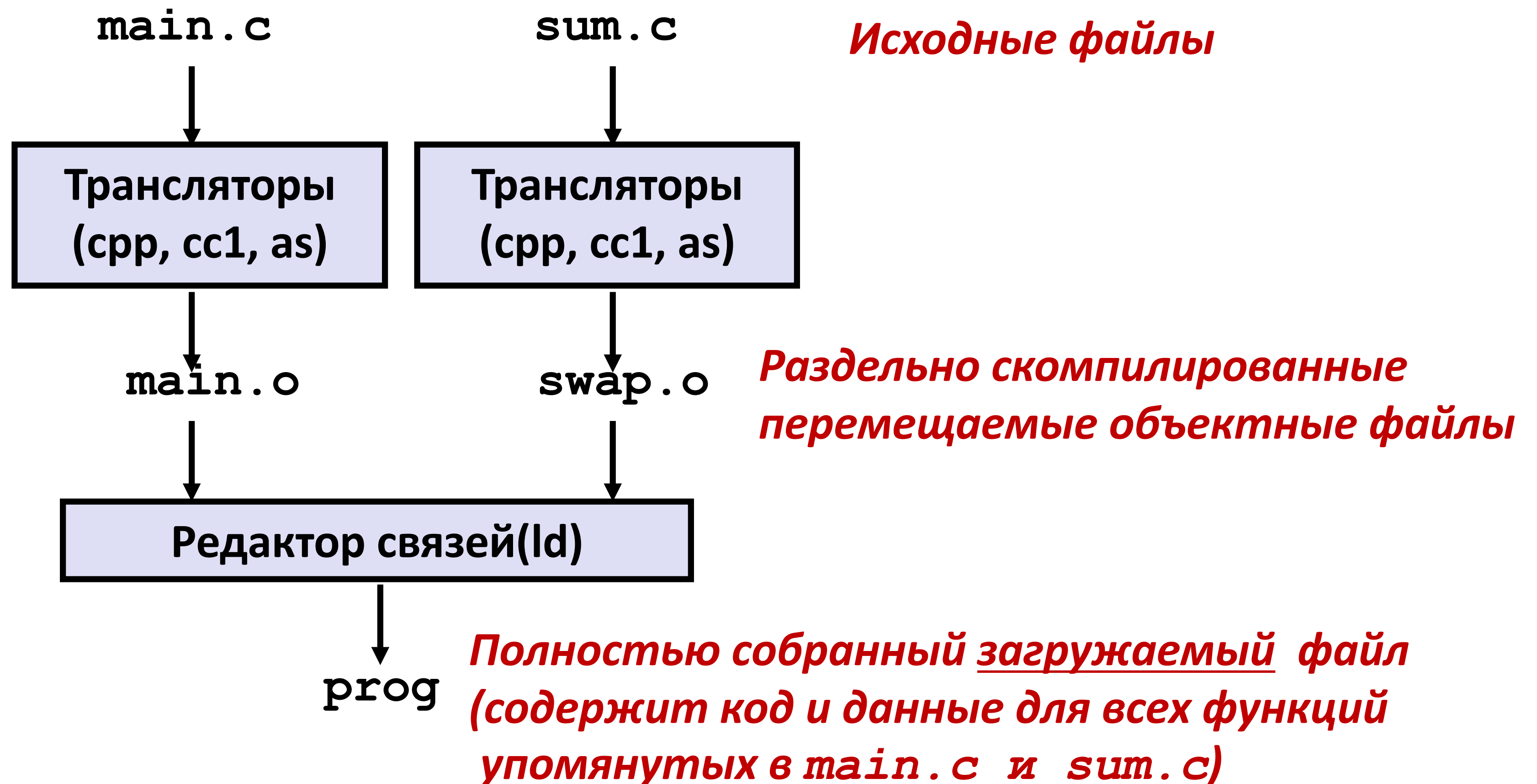
```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

sum.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

Статическое связывание

- **Трансляция и связывание *управляющей программой* :**
  - `unix> gcc -Og -o prog main.c sum.c`
  - `unix> ./prog`



Цели связывания

- **Модульность**
  - Программы могут создаваться не как один монолитный файл, а как набор небольших исходных файлов.
  - Возможно создание библиотек общих функций
    - например , библиотека math, стандартная библиотека Си

Цели связывания (продолжение)

- **Эффективность**
  - Временна́я: раздельная компиляция
    - перекомпиляция только изменённых исходных файлов
    - нет необходимости перекомпилировать другие исходные файлы
  - Пространственная: библиотеки
    - Общие функции могут быть собраны в общий файл...
    - При этом загружаемые файлы и исполняемая память содержат только код фактически используемых функций

Задачи связывания

- **Шаг 1: Разрешение символов**

  - В программах определяются и упоминаются *символы связывания* (переменные и функции):
    - `void swap() {…}    /* определяется символ swap */`
    - `swap();            /* упоминается символ swap */`
    - `int *xp = &x;      /* определяется xp, упоминается x */`

  - Определения символов сохраняются компилятором в *таблице символов*
    - Таблица символов – это массив записей-структур
    - Каждая запись включает имя, размер и положение символа.

  - **Редактор связей ставит в соответствие каждому упоминанию символа ровно одно определение этого символа**.

Задачи связывания (продолжение)

- **Шаг2: Перемещение кода**
  - Слияние множественных разделов кода и данных в единственные общие разделы
  - Перемещение символов из их относительных позиций, указанных в объектных (.o) файлах, в окончательные абсолютные позиции, указанные в загружаемом файле.
  - Редактирование всех ссылок на новое положение символов

Рассмотрим эти два шага подробнее ....

Три вида двоичных файлов (модулей)

- **Перемещаемые объектные файлы (`.o`)**
  - Содержат код и данные в форме, позволяющей комбинировать их с другими перемещаемыми объектными файлами при формировании исполняемого файла.
    - Каждый файл `.o` создаётся в точности из одного исходного (`.c`) файла
- **Загружаемые файлы (`a.out`)**
  - Содержат код и данные в форме, позволяющей непосредственно переносить их содержимое в память и затем исполнять.
- **Файлы переиспользуемых модулей (`.so`)**
  - Специальный тип перемещаемых объектных файлов, которые могут загружаться в память и связываться динамически во время загрузки и исполнения
  - В Windows называются *Dynamic Link Libraries* (DLLs)

Executable and Linkable Format (ELF)

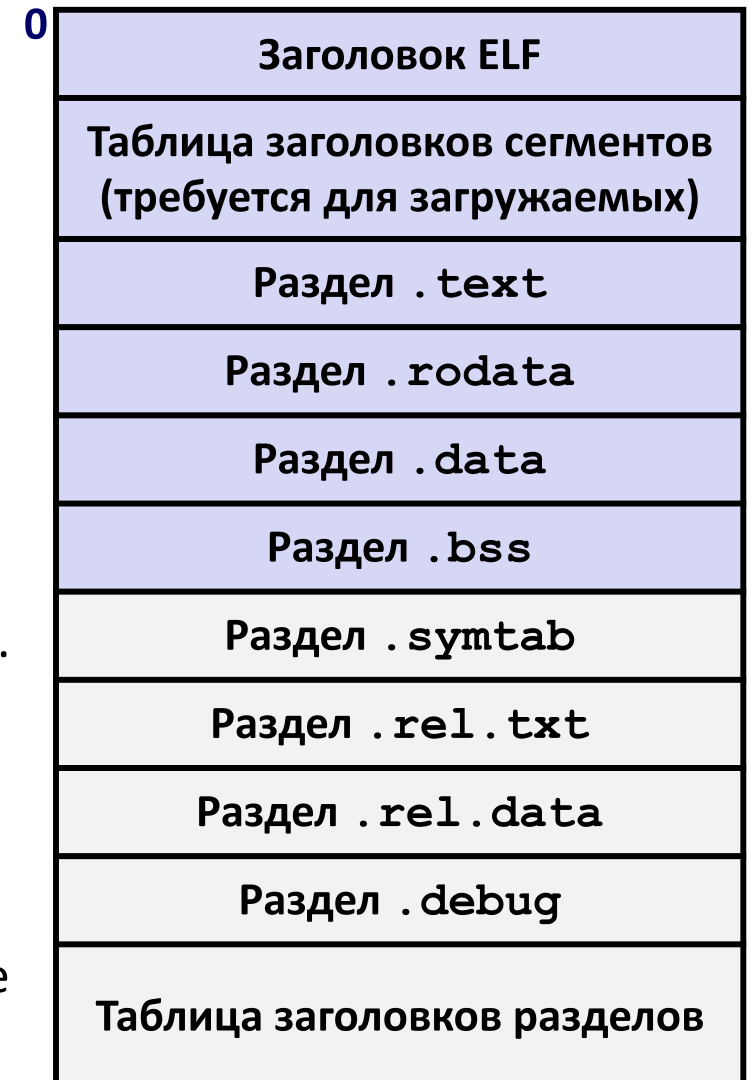
- **Стандартный двоичный формат для объектных файлов**

- **Один общий формат для**
  - перемещаемых объектных файлов (`.o`),
  - исполняемых файлов (`a.out`)
  - файлов переиспользуемых модулей (`.so`)

- **Общее название: ELF binaries**

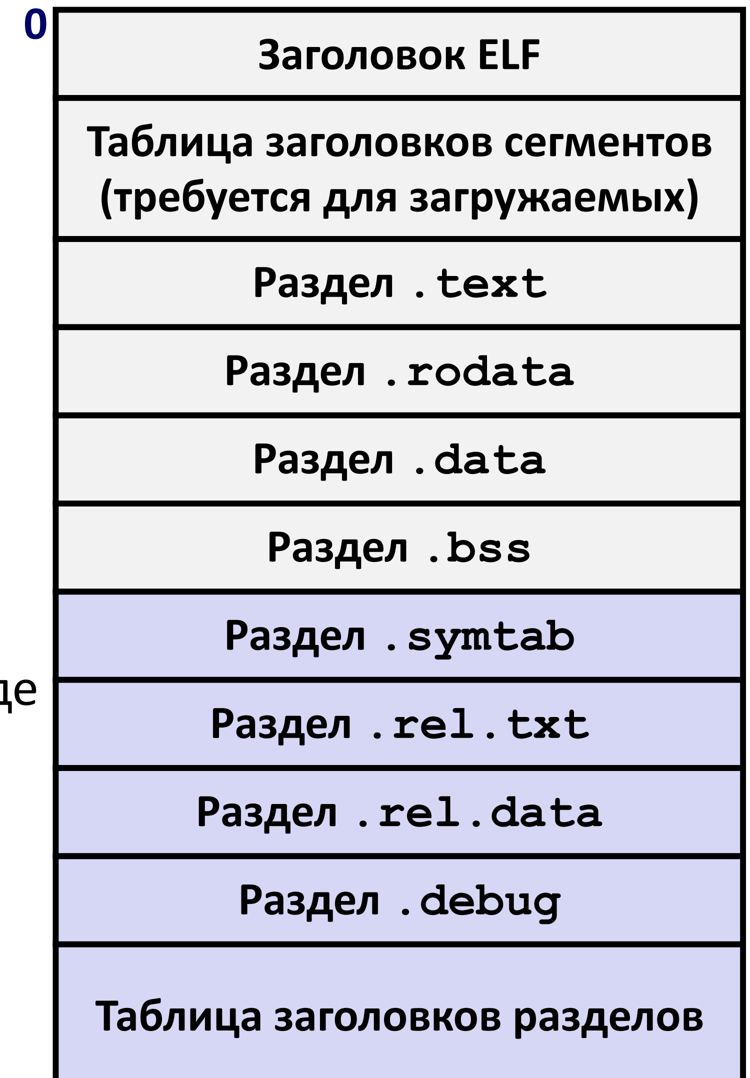
Объектный файл формата ELF

- **Заголовок Elf**
  - Размер слова, порядок байт, тип файла (.o, exec, .so), тип машины, и т.п.
- **Таблица заголовков сегментов памяти**
  - Размер страницы, сегменты виртуальных адресов памяти (разделы), размеры сегментов.
- **Раздел `.text`**
  - Код машинных команд
- **Раздел `.rodata`**
  - Данные только для чтения: таблицы переходов, ...
- **Раздел `.data`**
  - Инициализированные глобальные переменные
- **Раздел `.bss`**
  - Неинициализированные глобальные переменные
  - “Block Started by Symbol”, ”Better Save Space”
  - Имеет заголовок раздела, но не занимает места



Объектный файл формата ELF (продолж.)

- **Раздел `.symtab`**
  - Таблица символов
  - Имена процедур и статических переменных
  - Имена и размещения разделов
- **Раздел `.rel.text`**
  - Информация для перемещения раздела `.text`
  - Адреса команд, изменяемых в исполняемом коде
  - Модифицируемые команды
- **Раздел `.rel.data`**
  - Информация для перемещения раздела `.data`
  - Адреса указателей, изменяемых в исполняемом коде
- **Раздел `.debug`**
  - Информация для символьной отладки (`gcc -g`)
- **Таблица заголовков разделов**
  - Смещения и размеры каждого раздела



Связываемые символы

- **Глобальные символы**
  - Символы определённые данным модулем и упоминаемые другими
  - Примеры: не-**`static`** функции Си и не-**`static`** глобальные переменные

- **Внешние символы**
  - Глобальные символы упоминаемые данным модулем и определённые другими

- **Локальные символы**
  - Символы определяемые и упоминаемые только данным модулем
  - Примеры: функции и переменные Си определённые с атрибутом **`static`**
  - **Локальные связываемые символы *не есть* локальные переменные**

Шаг 1: Разрешение символов

Обращение к глобальному, …

… определённому здесь

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Определение глобального

Обращение к глобальному, …

Редактор связей ничего не знает о `val`

… определённому здесь

Редактор связей ничего не знает ни об `i`, ни об `s`

Локальные символы

- **Локальные переменные С хранятся…**
  - автоматические – в стеке (и они не символы редактора связей)
  - статические или в `.bss`, или в `.data` (и они символы редактора)

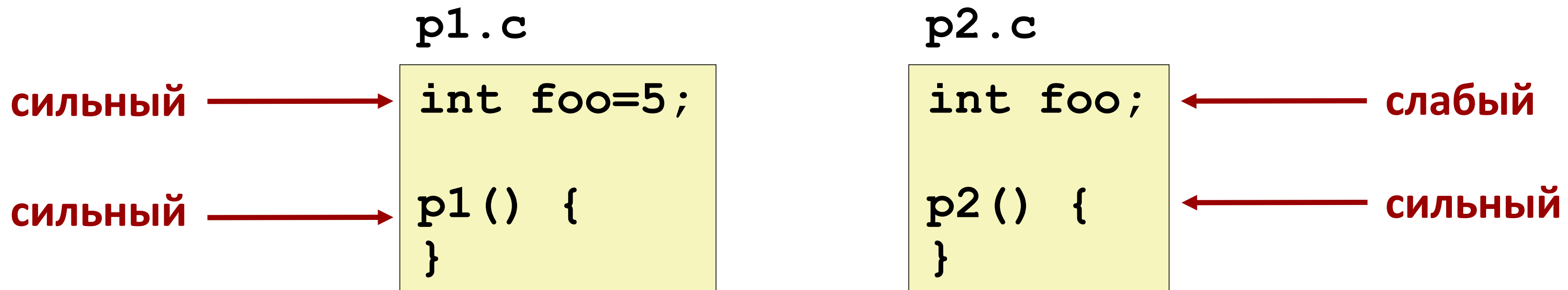
```c
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

Транслятор занимает место в `.data` для каждого их определений `x`

Создаёт локальные символы в таблице символов с уникальными именами, например, `x.1` и `x.2`.

Разрешение многократных определений символа

- **Символы программы…**
  - либо ***сильные***: процедуры и инициализированные глобальные
  - либо ***слабые***: неинициализированные глобальные



Правила связывания символов

- **Правило 1: не допускается несколько определений сильного символа**
  - Каждый символ определяется лишь однажды
  - Иначе: ошибка связывания

- **Правило 2: из нескольких определений символа выбирается сильное**
  - Определения слабых символов ассоциируются с сильным

- **Правило 3: из нескольких слабых определений символа выбирается любое**
  - Может отменяться с помощью `gcc –fno-common`

Головоломки связывания

Модуль 1	Модуль 2	Результат
`int x;` `p1() {}`	`p1() {}`	Ошибка связывания: два сильных символа (**`p1`**)
`int x;` `p1() {}`	`int x;` `p2() {}`	Упоминания **`x`** обратятся к одной целочисленной неинициализированной ячейке Это то, что вы хотели?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Запись в **`x`** в **`p2`** перезапишет **`y`** в **`p1`**!
		Беда!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Запись **`x`** в **`p2`** перезапишет **`y`** в **`p1`**!
		Опасность «изменения константы"!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	Упоминания **`x`** обратятся к одной целочисленной инициализированной ячейке.

Кошмарный случай: две одинаковых слабых структуры, компилированные с различными правилами выравнивания

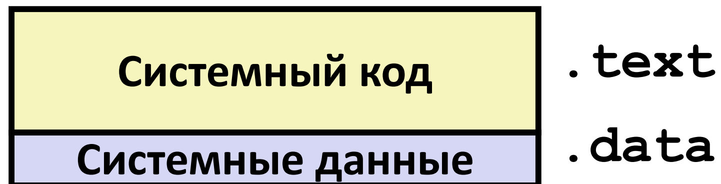
Глобальные переменные

- **Избегайте везде, где сможете**

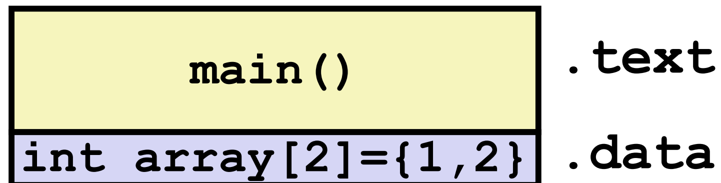
- **Иначе**
  - Используйте **`static`** везде, где сможете
  - Если используете глобальную, то инициализируйте её
  - Если обращаетесь к внешней глобальной, то используйте **`extern`**

Шаг 2: Перемещение кода и данных

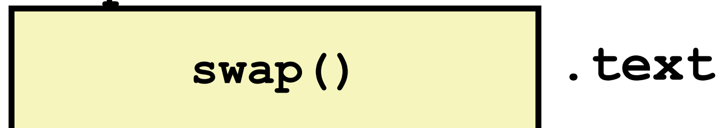
Файлы перемещаемых объектов



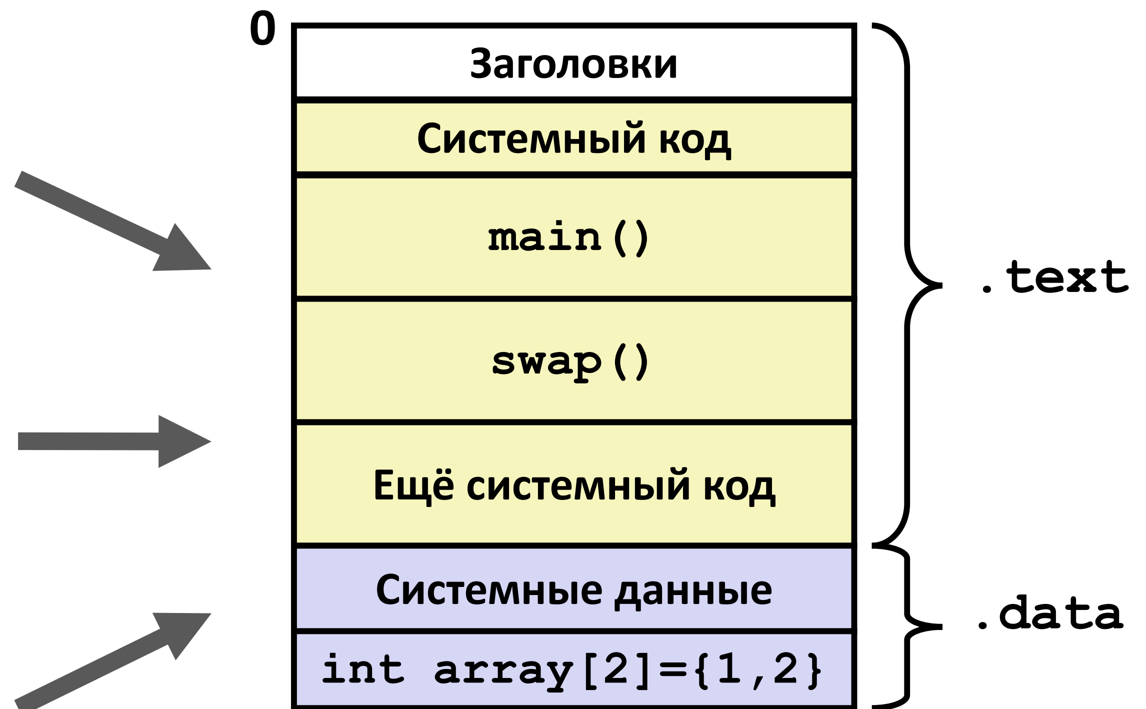
main.o



swap.o



Файл загружаемых объектов



Учёт перемещения

```c
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```
main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi      # %edi = &array
                        a: R_X86_64_32 array          # Учёт перемещения

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4      # Учёт перемещения
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```
main.o

Source: `objdump -r -d main.o`

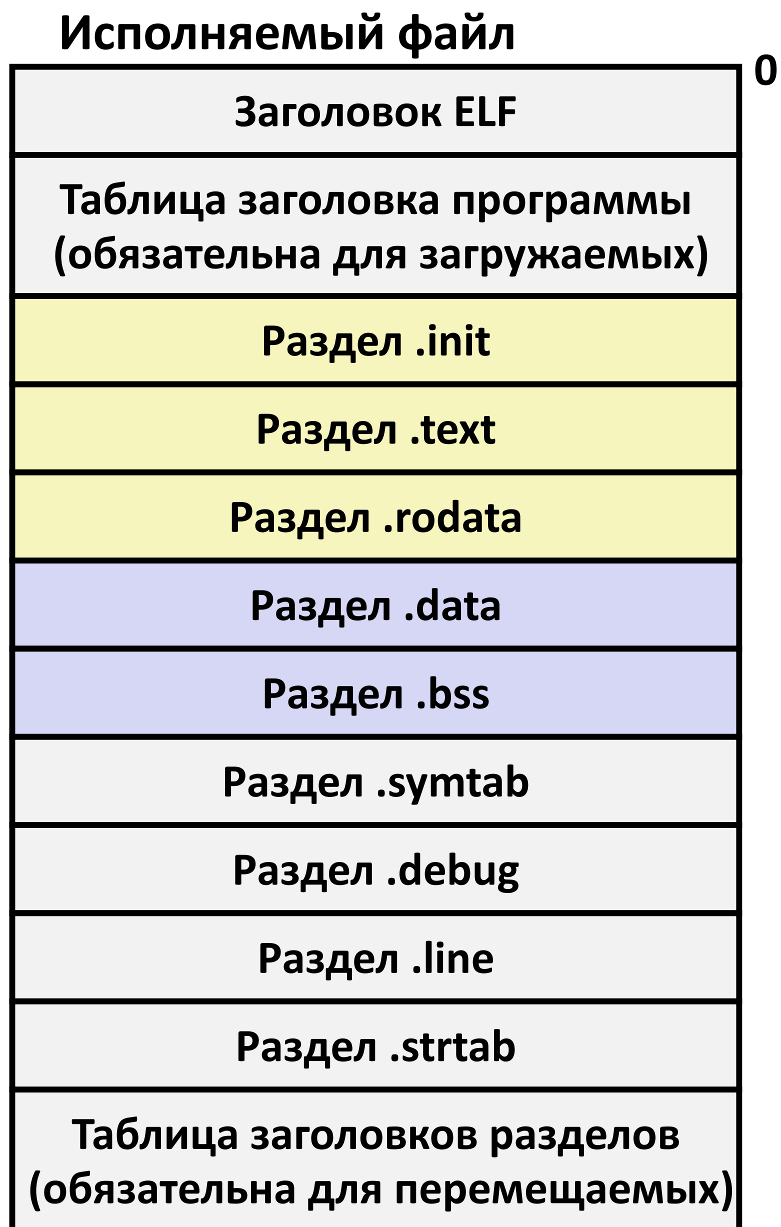
Перемещённый раздел .text

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00             mov    $0x0,%eax
  4004ed:       ba 00 00 00 00             mov    $0x0,%edx
  4004f2:       eb 09                      jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                   movslq %edx,%rcx
  4004f7:       03 04 8f                   add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                   add    $0x1,%edx
  4004fd:       39 f2                      cmp    %esi,%edx
  4004ff:       7c f3                      jl     4004f4 <sum+0xc>
  400501:       f3 c3                      repz retq
```

Для sum() используется адресация относительно PC : 0x4004e8 = 0x4004e3 + 0x5

Источник: `objdump -dx prog`

Загрузка загружаемых файлов



Память, недоступная прикладному коду

Виртуальная память ядра

Пользовательский стек (создан во время исполнения)

%rsp (стековый указатель)

Область отображения динамически связываемых библиотек

brk

«Куча» динамически распределяемой памяти (создаётся malloc)

Данные (чтение/запись) (.data, .bss)

Сегмент «только чтение» (.init, .text, .rodata)

Загружается из исполняемого файла

Не используется

0

Упаковка популярных функций

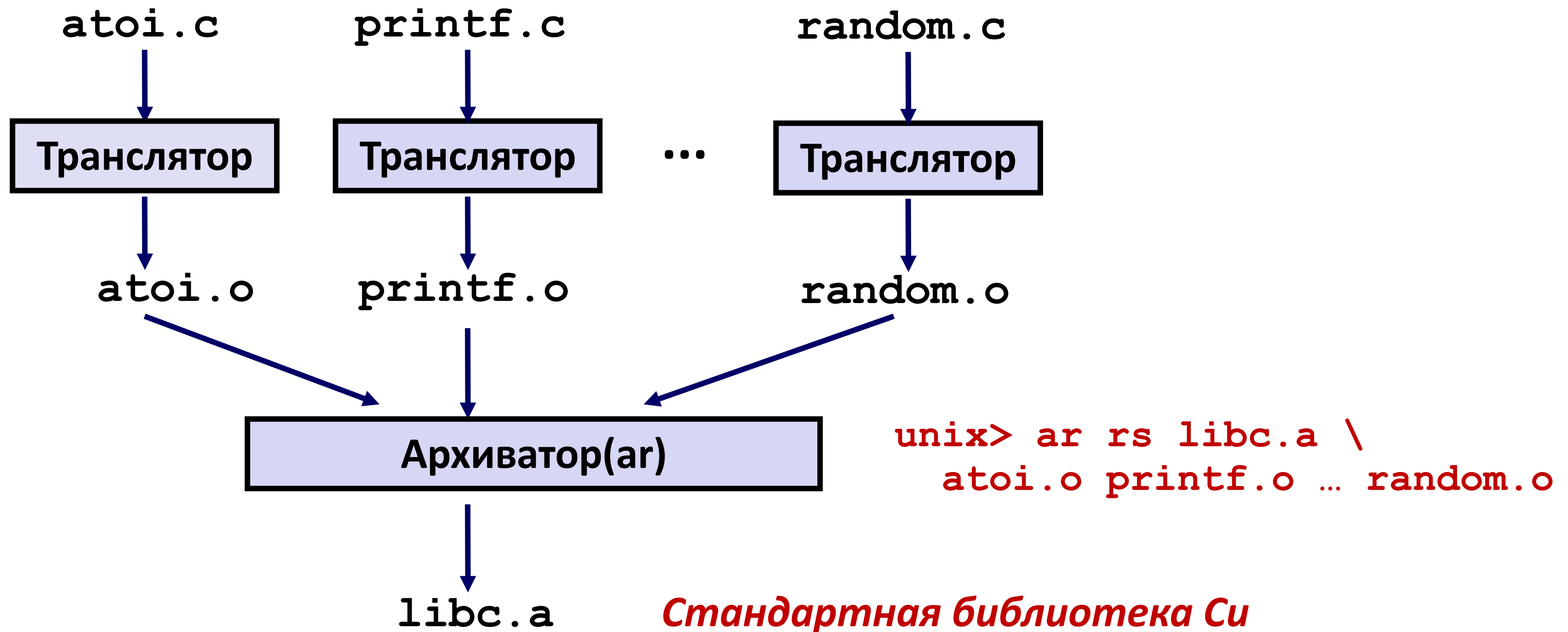
- **Как упаковать функции, обычно используемые программистами?**
  - Математика, ввод/вывод, управление памятью, работа со строками, и т.п.

- **Очевидные и неудобные возможности связывания:**
  - **Вариант 1:** Поместить все функции в один исходный файл
    - Программисты привязывают большой объект к своим программам
    - Неэффективное использование времени и пространства
  - **Вариант 2:** Поместить каждую функцию в отдельный исходный файл
    - Программист явно связывает соответствующие файлы со своей программой
    - Более эффективно, но затруднительно

Старомодное решение

- **Статические библиотеки (архивные файлы `.a`)**
  - Связанные перемещаемые объектные модули стыкуются в один файл с каталогом (*архив*).
  - Редактор связей расширяется так, чтобы разрешать неопределённые внешние ссылки в одном или нескольких архивах.
  - Если модуль из архива разрешает ссылку, то он связывается в исполняемый файл.

Создание статических библиотек



- Архиватор допускает помодульное изменение
- Перекомпиляция функции сопровождается заменой перемещаемого модуля в архиве

Популярные библиотеки

`libc.a` (стандартная библиотека Си)

- Архив размером 4.6 MB из 1496 объектных файлов.
- ввод/вывод, распределение памяти, обработка сигналов, обработка строк, дата и время, случайные числа, целочисленная математика

`libm.a` (математическая библиотека Си)

- Архив размером 2 MB из 444 объектного файла
- математика с плавающей точкой (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

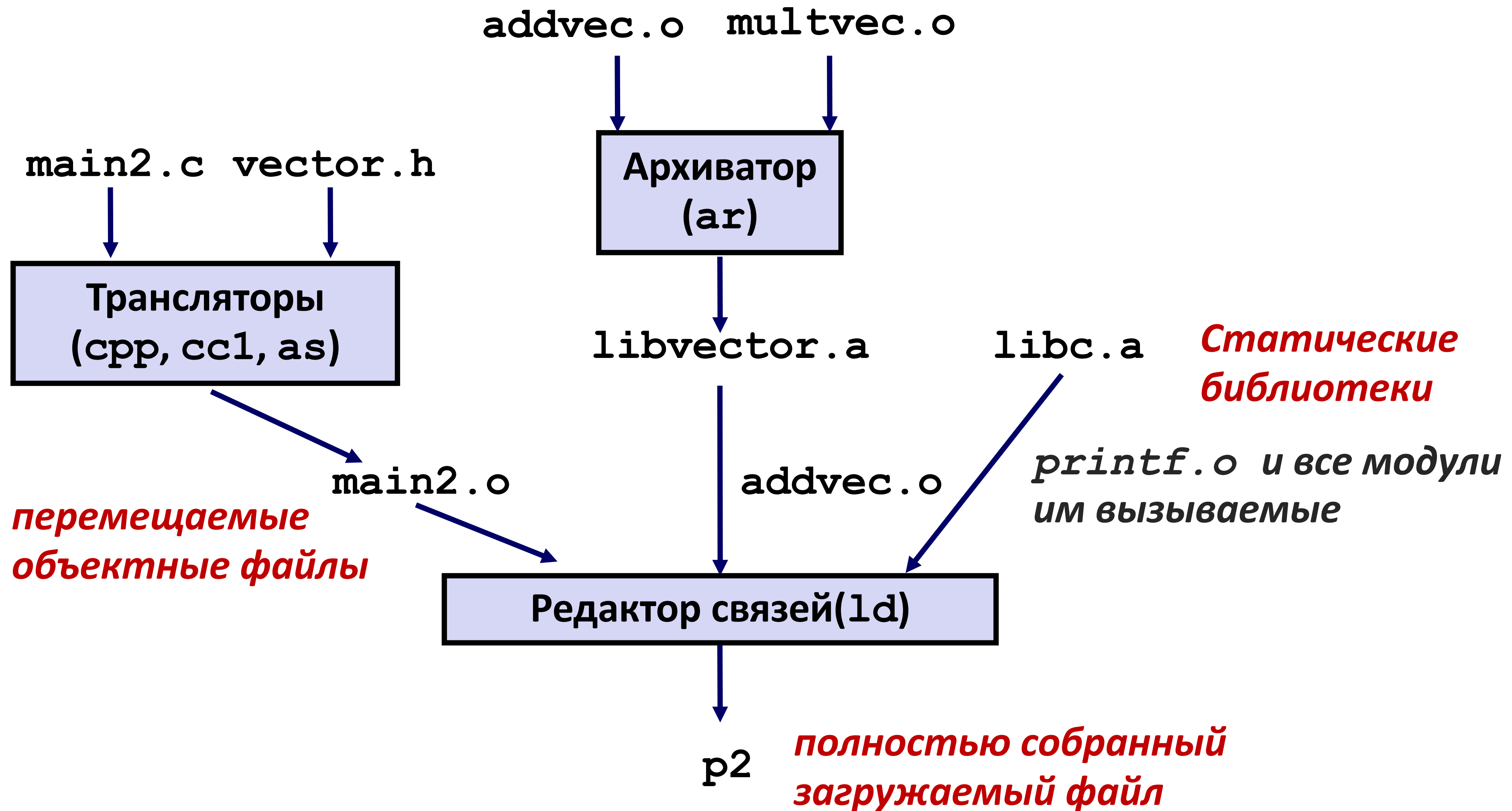
Связывание со статическими библиотеками

```
#include <stdio.h>
#include "vector.h"

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

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
           z[0], z[1]);
    return 0;
}
```
main2.c

```
int addcnt = 0;

void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    addcnt++;
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```
addvec.c

```
int multcnt = 0;

void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    multcnt++;
    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```
multvec.c

Связывание со статическими библиотеками



Использование статических библиотек

- **Алгоритм разрешения внешних ссылок редактором связей:**
  - Сканировать **.o** файлы и **.a** в порядке, указанном в командной строке.
  - Вести список неразрешённых ссылок.
  - В каждом новом **.o** или **.a** файле, пытаться разрешить каждую неразрешённую ссылку из списка символами определёнными в новом файле.
  - Если к завершению сканирования в списке неразрешённых останутся элементы, то выдать ошибку.

- **Проблема:**
  - Порядок указания в командной строке имеет значение!
  - Мораль: ставьте библиотеки в конце командной строки

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Современное решение

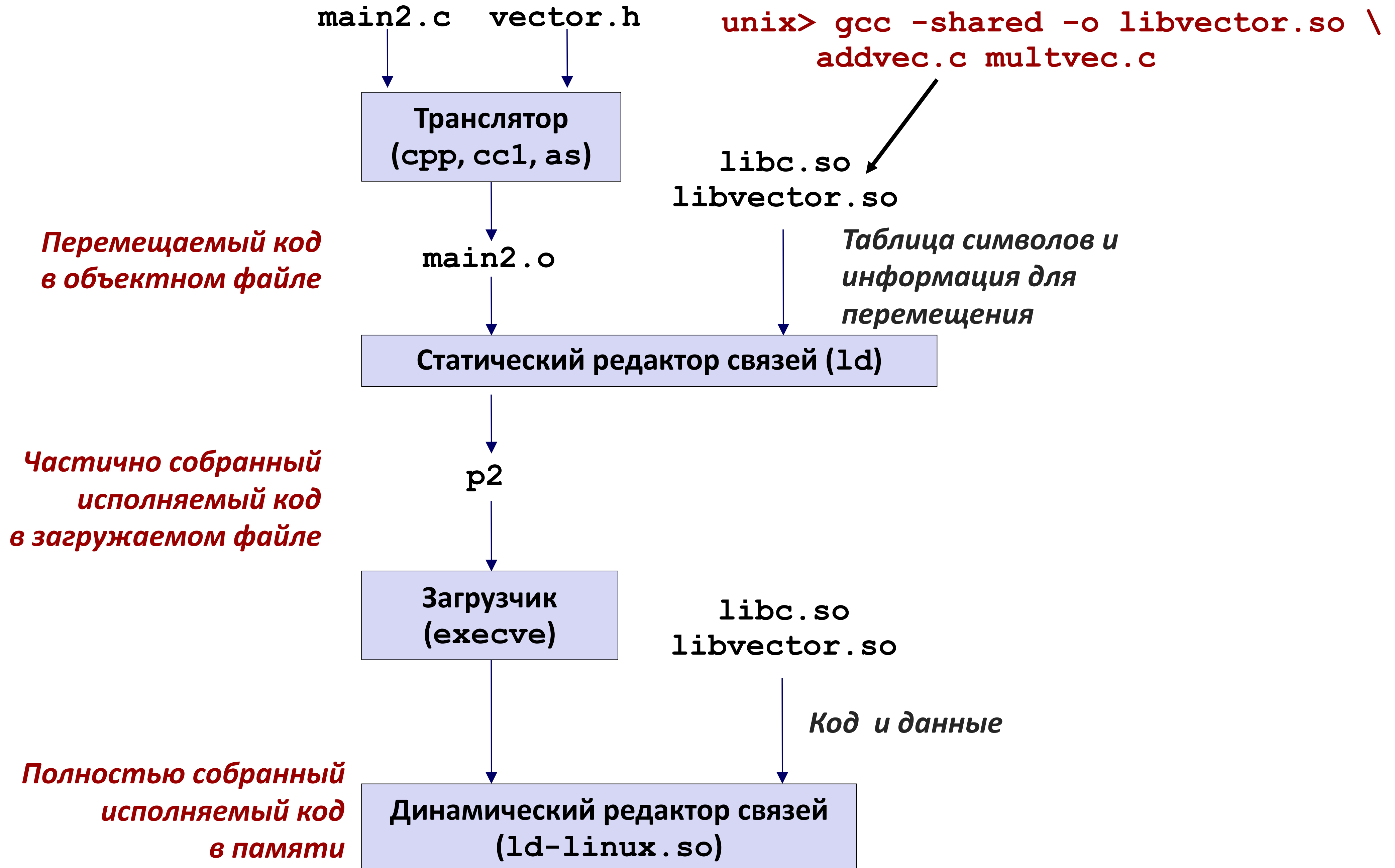
- **Недостатки статических библиотек:**
  - Тиражирование хранимого кода (каждой функции необходимы стандартная libc)
  - Тиражирование исполняемого кода
  - Даже небольшие исправления в системных библиотеках требуют явной пересборки каждого приложения

- **Разделяемые библиотеки (Shared Libraries)**
  - Файлы содержащие код и данные, загружаемые и связываемые к приложением *динамически,* либо во *время загрузки*, либо во *время исполнения*
  - Другие названия: динамически связываемые бибилиотеки, DLL, `.so` файлы

Современное решение (продолжение)

- **Динамическое связывание может производиться во время загрузки и начала исполнения.**
  - Обычный случай для Linux, выполняется автоматически динамическим редактором связей (**`ld-linux.so`**).
  - Стандартная библиотека Си (**`libc.so`**) связывается динамически.
- **Динамическое связывание может производиться во время исполнения.**
  - В Linux, это выполняется вызовами интерфейса **`dlopen()`**
    - Распределённое ПО
    - Высокопроизводительные веб-сервера
    - Посредничество при вызове процедур во время исполнения.
- **Код и данные разделяемых библиотек могут переиспользоваться несколькими процессами**
  - С использованием механизмов виртуальной памяти

Динамическое связывание при загрузке



Динамическое связывание при исполнении

```c
#include <stdio.h>
#include <stdlib.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;

    /* Динамическая загрузка библиотеки содержащей addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c

Динамическое связывание при исполнении

```c
    ...

    /* взятие указателя на только что загруженную addvec() */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

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

    /* выгрузка разделяемой библиотеки */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Связывание: сводка

- **Связывание позволяет компоновать программы из множества объектных файлов**

- **Связывание происходит в различные моменты времени жизни программы:**
  - во время компиляции
  - во время загрузки
  - во время исполнения

- **Понимание связывания помогает избежать противных ошибок и стать лучшим, чем раньше, программистом**

Пример: «библиотечное опосредование»

- **Library interpositioning - мощная техника связывания. Позволяет программисту перехватывать вызовы произвольных функций**
- **Опосредование может применяться во время…**

  …компиляции исходного кода,

  … статического связывания перемещаемых объектных файлов в исполняемый файл,

  …загрузки исполняемого файла в память, динамического связывания и исполнения

Некоторые применения опосредования

- **Безопасность**
  - Безопасная среда (песочница)
  - Фоновое шифрование
- **Отладка**
  - В 2014 два инженера Facebook , используя опосредование, устранили предательскую, годовой давности ошибку в приложении для iPhone
  - Код сетевого стека SPDY делал запись в ошибочное место
  - Помог перехват обращений к функциям записи Posix (write, writev, pwrite)

Источник: сообщение в блоге разработки Facebook

https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

Некоторые применения опосредования (продолжение)

- **Наблюдение и профилирование**
  - Подсчёт количества вызовов функций
  - Определение мест и аргументов вызова функций
  - Трассировка malloc
    - Обнаружение утечек памяти
    - **Трассировка адресов**

Пример программы

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

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```

int.c

- **Цель: трассировка адресов и размеров выделяемых и освобождаемых блоков, без изменения исходного кода.**

- **Три решения: опосредование функций `malloc` и `free` при компиляции, статическом связывании и загрузке/исполнении.**

Опосредование компиляции

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* обёртка функции malloc */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n",
           (int)size, ptr);
    return ptr;
}

/* обёртка функции free */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Опосредование при компиляции

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```

Опосредование статического связывания

```
#ifdef LINKTIME
#include <stdio.h>

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

/* обёртка функции malloc */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* обёртка функции free */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Опосредование статического связывания

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl
int.o mymalloc.o
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```

- **Флаг “`-Wl`” передаёт аргумент редактору связей, заменяя каждую запятую пробелом.**
- **“`--wrap,malloc`” командует редактору связей разрешать упоминания специальным образом:**
  - Упоминание `malloc` должно разрешаться как `__wrap_malloc`
  - Упоминание `__real_malloc` должно разрешаться как `malloc`

Опосредование при загрузке/исполнении

```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* обёртка функции malloc */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

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

mymalloc.c

Опосредование при загрузке/исполнении

```
/* обёртка функции free */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* получение адреса libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Вызов libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Опосредование при загрузке/исполнении

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```

- **Переменная среды `LD_PRELOAD` указывает динамическому редактору связей сначала разрешать упоминания (например для `malloc`) в `mymalloc.so`**

Опосредование: сводка

- **Во время компиляции**
  - Видимые вызовы malloc/free с помощью макрорасширений преобразуются в вызовы mymalloc/myfree
- **Во время статического связывания**
  - Используется трюк редактора связей для специального разрешения упоминаний
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- **Во время загрузки/исполнения**
  - Реализует специальную версию malloc/free, которая использует динамическое связывание для загрузки библиотечных malloc/free с другими именами