

Связывание

Основы информатики

Компьютерные основы программирования

`u.to/DbCmFA`

На основе CMU 15-213/18-243:

Introduction to Computer Systems

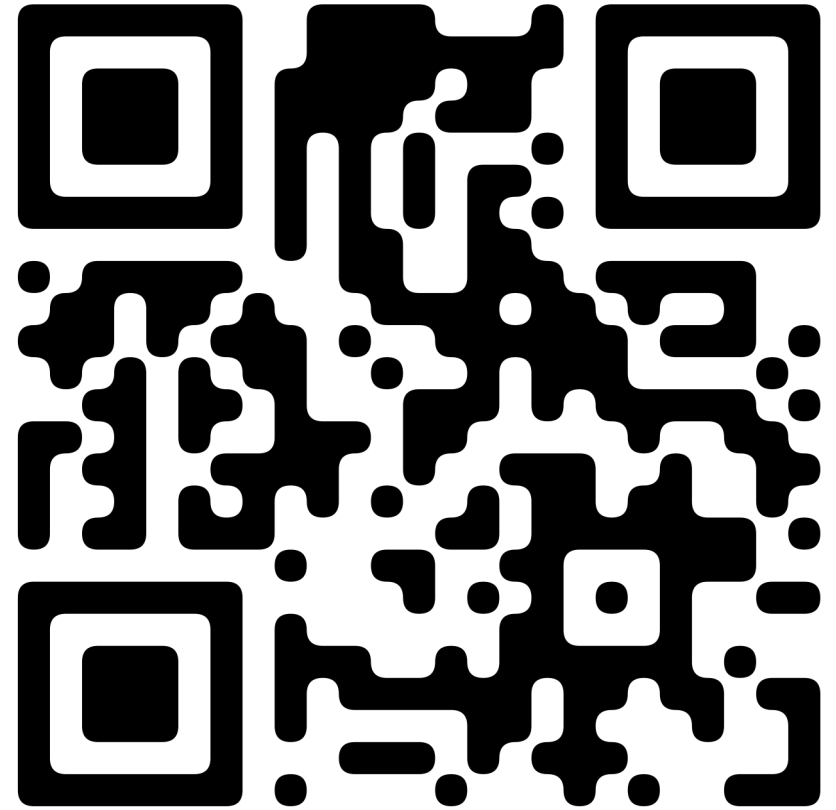
`u.to/XoKmFA`

Лекция 11, 21 апреля 2023

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

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



Пример Си-программы

`main.c`

```
int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

`swap.c`

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

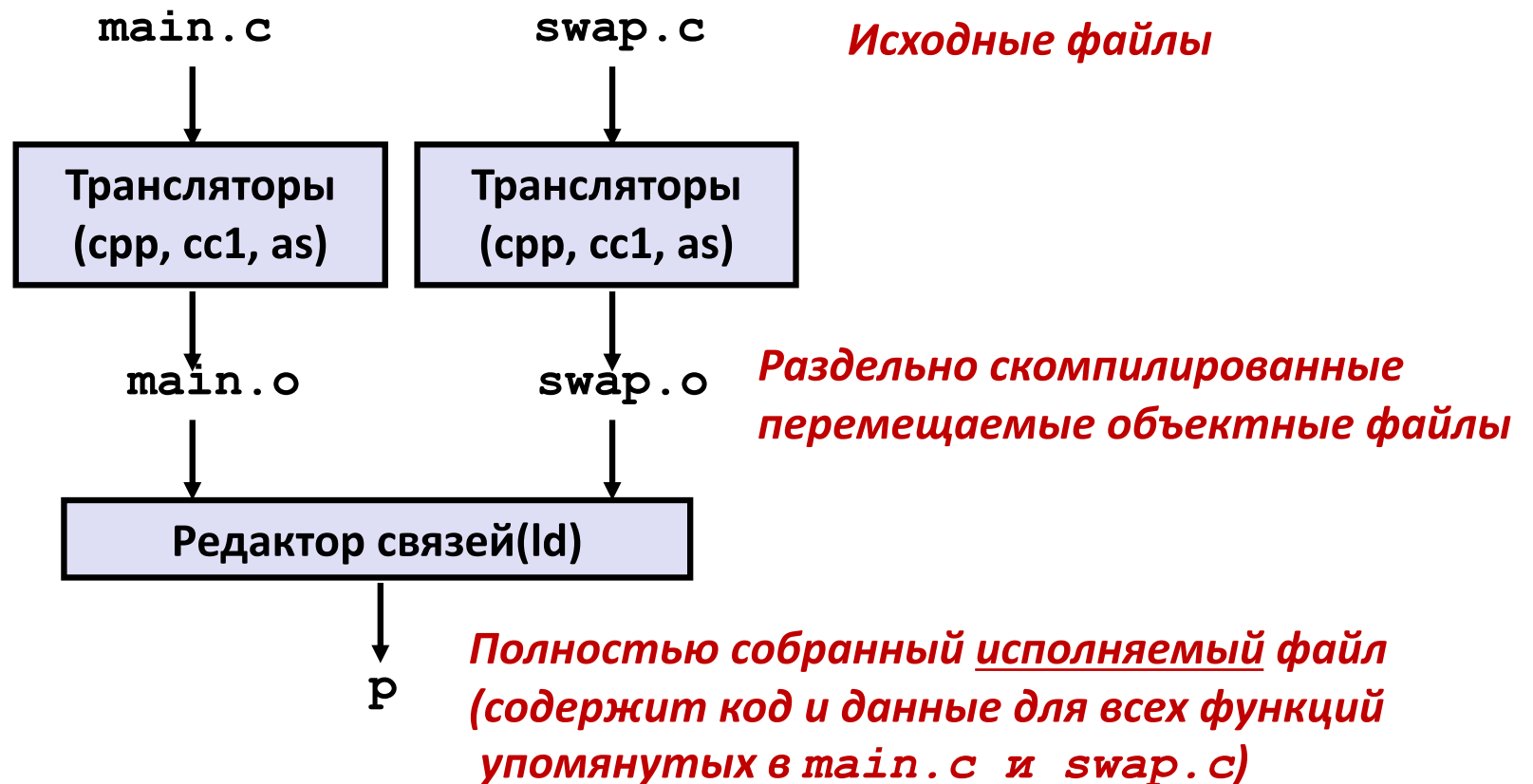
void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Статическое связывание

■ Трансляция и связывание *управляющей программой* :

- `unix> gcc -O2 -g -o p main.c swap.c`
- `unix> ./p`



Цели связывания – 1

■ 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 – 2

■ Эффективность

- Время: отдельная компиляция
 - изменение одного исходного файла, его компиляция и пересборка
 - нет необходимости перекомпилировать другие исходные файлы
- Пространство: библиотеки
 - Общие функции могут быть собраны в общий файл...
 - При этом исполняемые файлы и исполняющийся фрагмент памяти содержат только код фактически используемых функций

Задачи связывания - 1

■ Разрешение символов

- В программах определяются и упоминаются *символы* (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 – 2

■ 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 исполняемом файле.
- Редактирование всех ссылок на символы чтобы указать новое положение.

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

■ Исполняемые файлы (a . out)

- Содержат код и данные в форме, позволяющей непосредственно копировать 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**
- **Изначально предложен в AT&T System V Unix**
 - Позже адаптирован для вариантов BSD Unix и Linux
- **Один общий формат для**
 - перемещаемых объектных файлов (`.o`),
 - исполняемых файлов (`a.out`)
 - файлов переиспользуемых модулей (`.so`)
- **Общее название: ELF binaries**

Объектный файл формата ELF - 1

■ Заголовок Elf

- Размер слова, порядок байт, тип файла (.o, exes, .so), тип машины, и т.п.

■ Таблица заголовков сегментов

- 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”
- Имеет заголовок раздела, но не занимает места

0	Заголовок ELF
	Таблица заголовков сегментов (требуется для исполняемых)
	Раздел .text
	Раздел .rodata
	Раздел .data
	Раздел .bss
	Раздел .symtab
	Раздел .rel.txt
	Раздел .rel.data
	Раздел .debug
	Таблица заголовков разделов

Объектный файл формата ELF - 2

■ 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

0	Заголовок ELF
	Таблица заголовков сегментов (требуется для исполняемых)
	Раздел <code>.text</code>
	Раздел <code>.rodata</code>
	Раздел <code>.data</code>
	Раздел <code>.bss</code>
	Раздел <code>.symtab</code>
	Раздел <code>.rel.txt</code>
	Раздел <code>.rel.data</code>
	Раздел <code>.debug</code>
	Таблица заголовков разделов

Связываемые символы

■ Глобальные символы

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

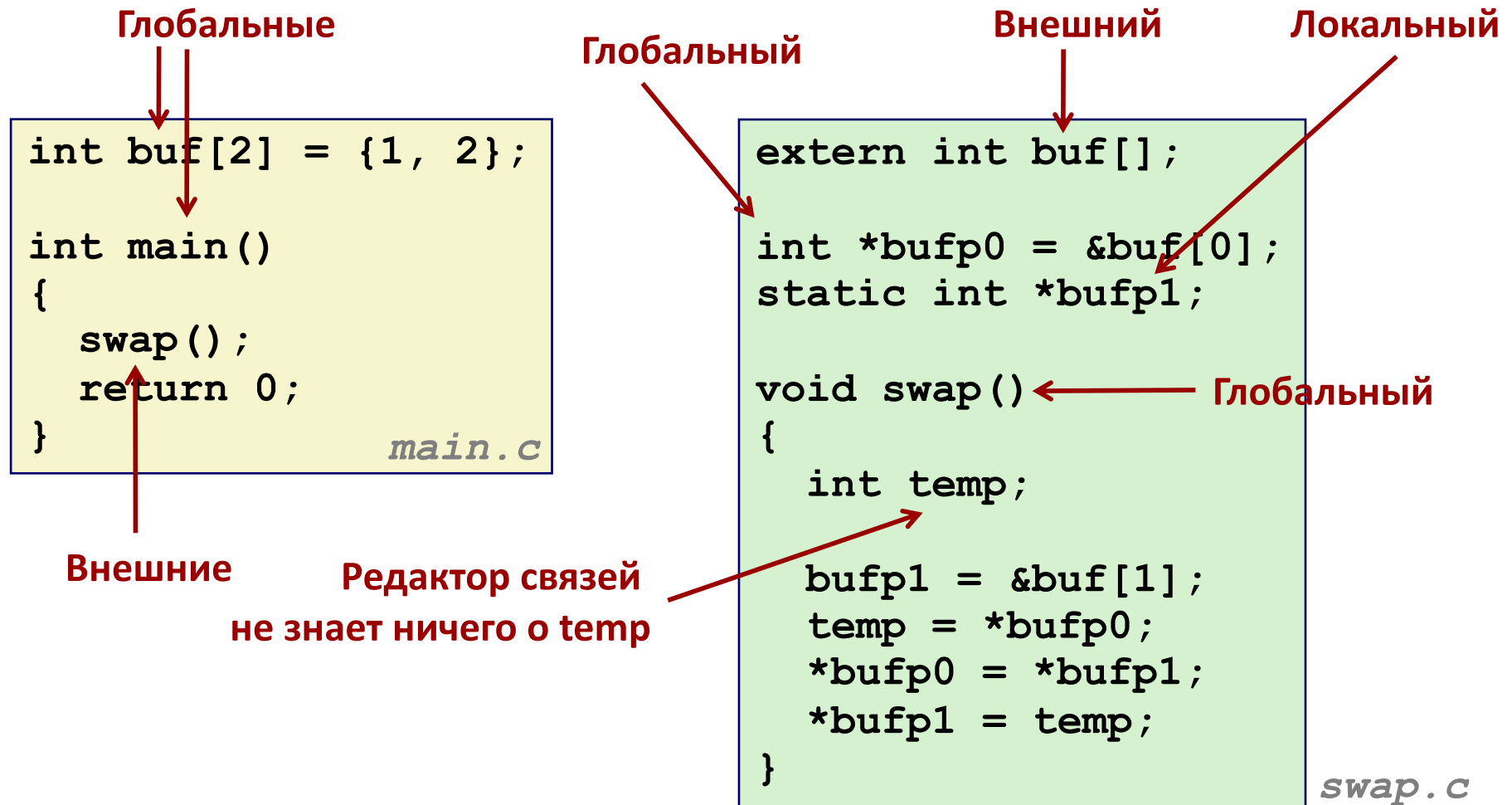
■ Внешние символы

- Глобальные символы упоминаемые данным модулем и определённые другими

■ Локальные символы

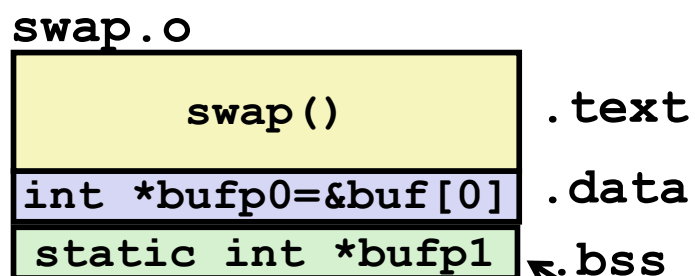
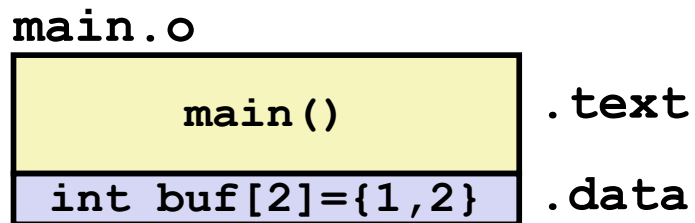
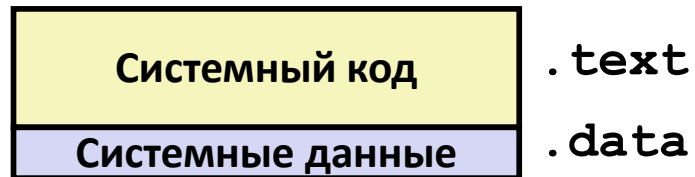
- Символы определяемые и упоминаемые только данным модулем
- Примеры: функции и переменные Си определённые с атрибутом **static**
- **Локальные связываемые символы не есть локальные переменные**

Разрешение символов

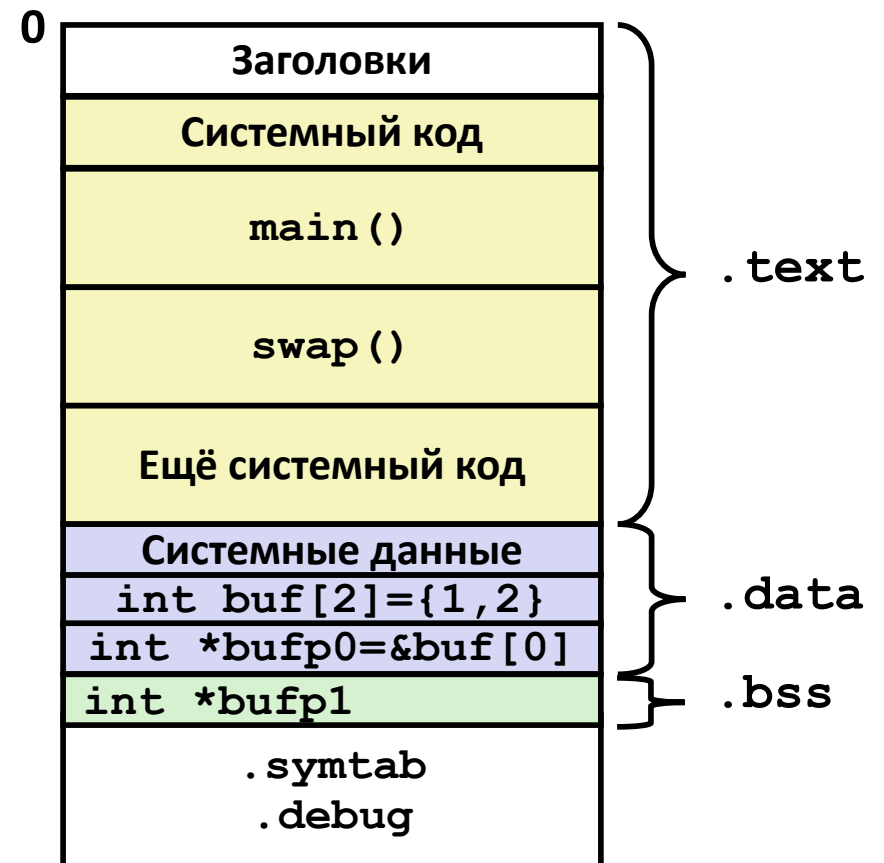


Перемещение кода и данных

Файлы перемещаемых объектов



Файл исполняемых объектов



Хотя и только для swap, требуется размещение в .bss

Информация для перемещения (main)

main.c

```
int buf[2] =  
    {1,2};  
  
int main()  
{  
    swap();  
    return 0;  
}
```

main.o

```
00000000 <main>:  
  0:  8d 4c 24 04      lea     0x4(%esp),%ecx  
  4:  83 e4 f0         and     $0xffffffff0,%esp  
  7:  ff 71 fc         pushl   0xfffffffffc(%ecx)  
  a:  55              push    %ebp  
  b:  89 e5           mov     %esp,%ebp  
  d:  51             push    %ecx  
  e:  83 ec 04        sub     $0x4,%esp  
11:  e8 fc ff ff ff   call    12 <main+0x12>  
                        12: R_386_PC32 swap  
16:  83 c4 04        add     $0x4,%esp  
19:  31 c0          xor     %eax,%eax  
1b:  59             pop     %ecx  
1c:  5d             pop     %ebp  
1d:  8d 61 fc        lea     0xfffffffffc(%ecx),%esp  
20:  c3             ret
```

Дизассемблирование секции .data:

```
00000000 <buf>:  
  0:  01 00 00 00 02 00 00 00
```

Источник: `objdump -r -d`

Инф. для перемещения (swap, .text)

swap.c

```
extern int buf[];

int
    *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

swap.o

Дизассемблирование раздела .text:

00000000 <swap>:

0:	8b 15 00 00 00 00	mov	0x0,%edx
	2: R_386_32	buf	
6:	a1 04 00 00 00	mov	0x4,%eax
	7: R_386_32	buf	
b:	55	push	%ebp
c:	89 e5	mov	%esp,%ebp
e:	c7 05 00 00 00 00 04	movl	\$0x4,0x0
15:	00 00 00		
	10: R_386_32	.bss	
	14: R_386_32	buf	
18:	8b 08	mov	(%eax),%ecx
1a:	89 10	mov	%edx,(%eax)
1c:	5d	pop	%ebp
1d:	89 0d 04 00 00 00	mov	%ecx,0x4
	1f: R_386_32	buf	
23:	c3	ret	

Инф. для перемещения (swap, .data)

swap.c

```
extern int buf[];

int *bufp0 =
    &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Дизассемблирование раздела .data:

```
00000000 <bufp0>:
    0:  00 00 00 00

    0:  R_386_32 buf
```

Исполняемый до/после перемещ. (.text)

00000000 <main>:

```

    . . .
    e:  83 ec 04          sub    $0x4,%esp
   11:  e8 fc ff ff ff    call   12 <main+0x12>
                        12: R_386_PC32 swap
   16:  83 c4 04          add    $0x4,%esp
    . . .
```

0x8048396 + 0x1a
= 0x80483b0

08048380 <main>:

```

8048380:      8d 4c 24 04          lea    0x4(%esp),%ecx
8048384:      83 e4 f0          and    $0xfffffffff0,%esp
8048387:      ff 71 fc          pushl  0xfffffffffc(%ecx)
804838a:      55                push   %ebp
804838b:      89 e5                mov    %esp,%ebp
804838d:      51                push   %ecx
804838e:      83 ec 04          sub    $0x4,%esp
8048391:      e8 1a 00 00 00    call   80483b0 <swap>
8048396:      83 c4 04          add    $0x4,%esp
8048399:      31 c0          xor    %eax,%eax
804839b:      59                pop    %ecx
804839c:      5d                pop    %ebp
804839d:      8d 61 fc          lea    0xfffffffffc(%ecx),%esp
80483a0:      c3                ret
```

```

0:  8b 15 00 00 00 00      mov     0x0,%edx
           2: R_386_32      buf
6:  a1 04 00 00 00      mov     0x4,%eax
           7: R_386_32      buf
...
e:  c7 05 00 00 00 00 04  movl    $0x4,0x0
15:  00 00 00
           10: R_386_32      .bss
           14: R_386_32      buf
. . .
1d:  89 0d 04 00 00 00      mov     %ecx,0x4
           1f: R_386_32      buf
23:  c3                      ret

```

080483b0 <swap>:

```

80483b0:  8b 15 20 96 04 08      mov     0x8049620,%edx
80483b6:  a1 24 96 04 08      mov     0x8049624,%eax
80483bb:  55                      push    %ebp
80483bc:  89 e5                  mov     %esp,%ebp
80483be:  c7 05 30 96 04 08 24  movl    $0x8049624,0x8049630
80483c5:  96 04 08
80483c8:  8b 08                  mov     (%eax),%ecx
80483ca:  89 10                  mov     %edx,(%eax)
80483cc:  5d                      pop     %ebp
80483cd:  89 0d 24 96 04 08      mov     %ecx,0x8049624
80483d3:  c3                      ret

```

Исполняемый после перемещения (.data)

Дизассемблирование раздела .data:

08049620 <buf>:

8049620: 01 00 00 00 02 00 00 00

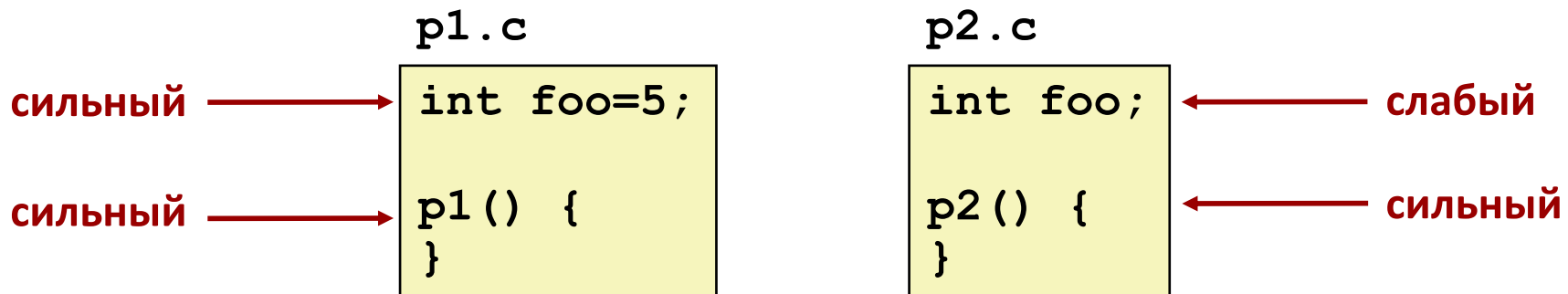
08049628 <bufp0>:

8049628: 20 96 04 08

Сильные и слабые символы

■ Символы программы...

- либо **сильные**: процедуры и инициализированные глобальные
- либо **слабые**: неинициализированные глобальные



Правила разрешения символов

- **Правило 1: не допускаются 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

```
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!

```
int x=7;  
p1() {}
```

```
int x;  
p2() {}
```

Упоминания **x** обратятся к одной целочисленной
инициализированной ячейке.

Кошмарный случай: две одноимённых слабых структуры, скомпилированные разными компиляторами с различными правилами выравнивания

Роль .h файлов – 1

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

c2.c

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

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

global.h

```
#ifndef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```


Запуск препроцессора

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

-DINITIALIZE

без инициализации

```
int g = 23;
static int init = 1;
int f() {
    return g+1;
}
```

```
int g;
static int init = 0;
int f() {
    return g+1;
}
```

#include заставляет препроцессор Си вставить файл буквально

Роль .h файлов – 2

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
#ifndef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c2.c

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

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

Что получится:

```
gcc -o p c1.c c2.c
??
```

```
gcc -o p c1.c c2.c \
-DINITIALIZE
??
```

Глобальные переменные

- Избегайте, где только можно
- Иначе
 - Используйте **static** если возможно
 - Инициализируйте, если вы используете глобальную переменную
 - Используйте **extern** если вы обращаетесь к внешней глобальной переменной

Упаковка популярных функций

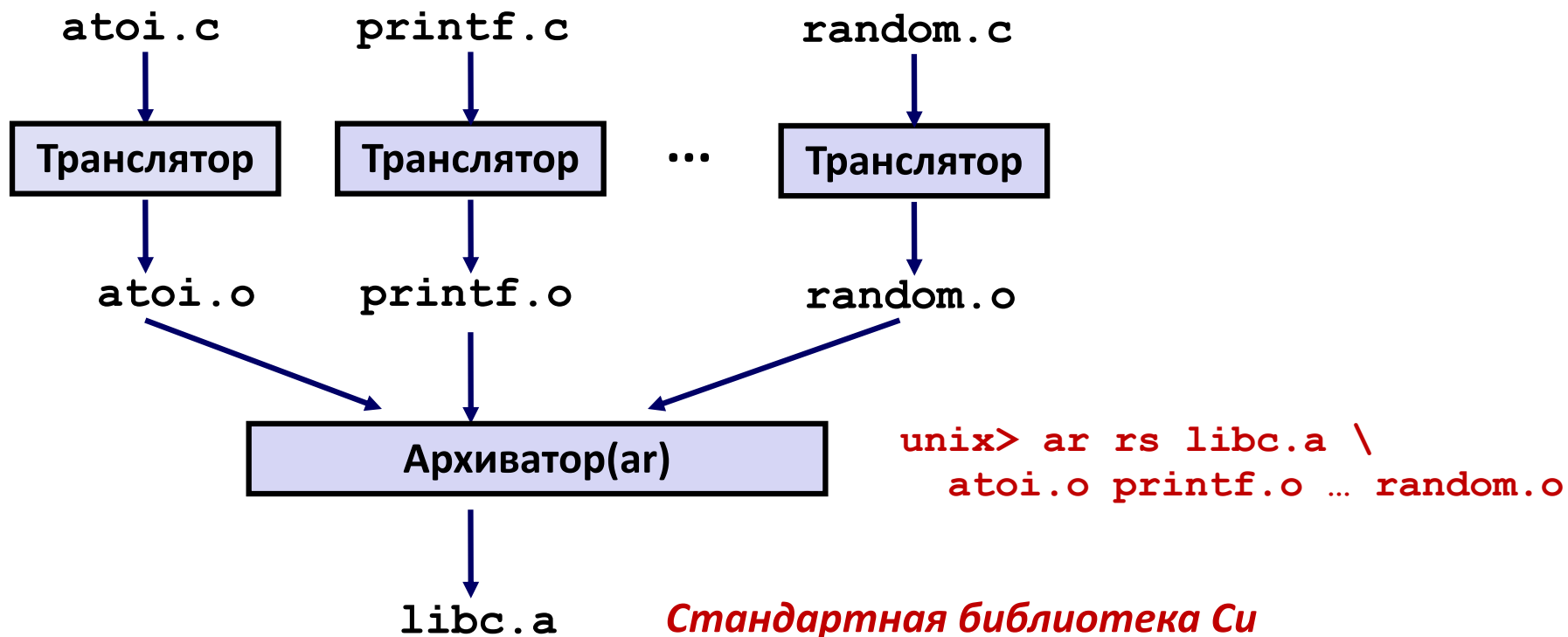
- Как упаковать функции, обычно используемые программистами?
 - Математика, ввод/вывод, управление памятью, работа со строками, и т.п.
- Очевидные и неудобные возможности связывания:
 - **Вариант 1:** Поместить все функции в один исходный файл
 - Программисты привязывают большой объект к своим программам
 - 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

Решение: статические библиотеки

■ Статические библиотеки (архивные файлы .a)

- Связанные перемещаемые объектные модули стыкуются в один файл с индексом (*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 изменением и заменой объектного файла в архиве.

Популярные библиотеки

libc.a (стандартная библиотека Си)

- Архив размером 8 МВ из 1392 объектных файлов.
- ввод/вывод, распределение памяти, обработка сигналов, обработка строк, дата и время, случайные числа, целосисленная математика

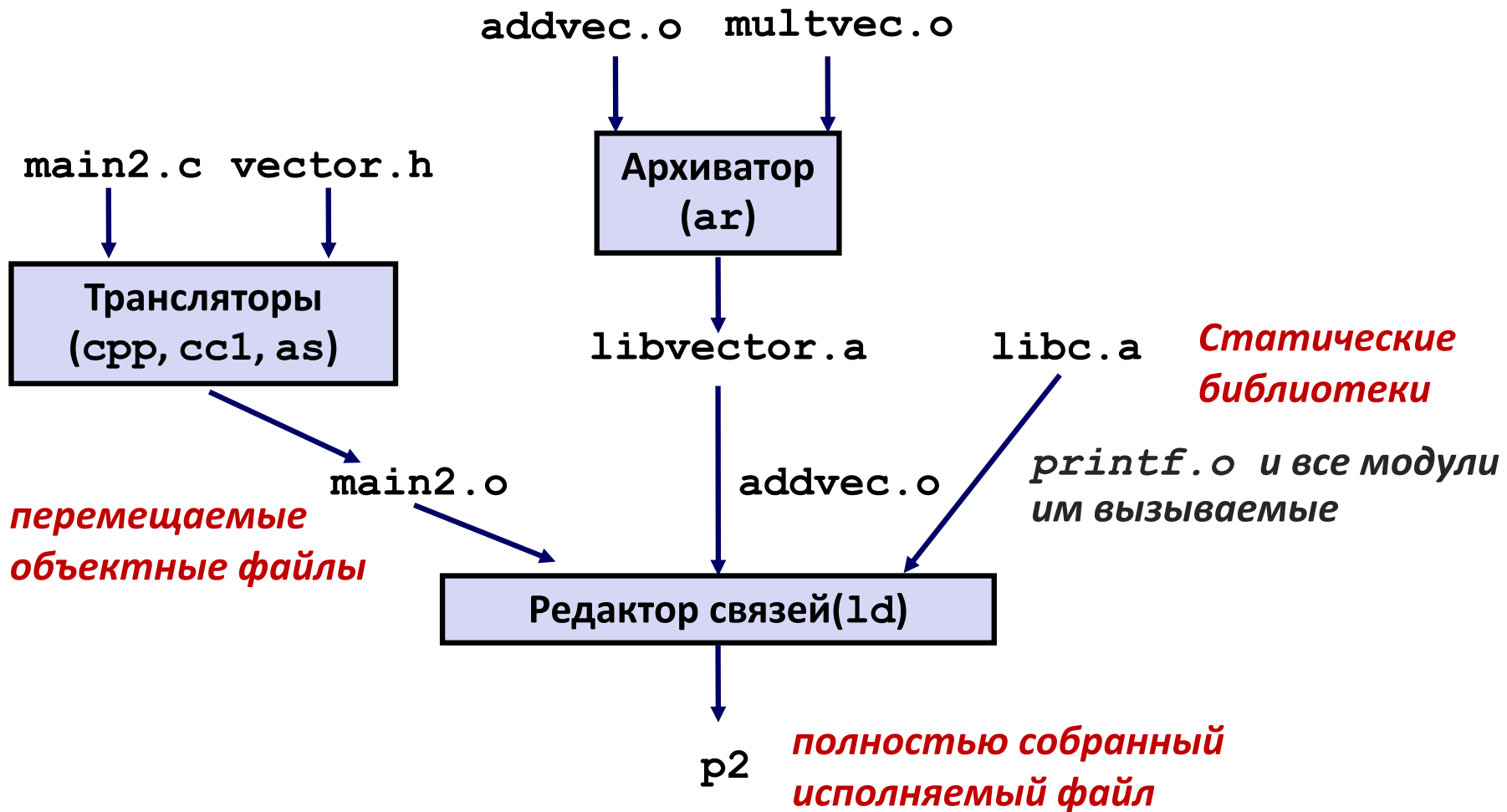
libm.a (математическая библиотека Си)

- Архив размером 1 МВ из 401 объектного файла
- математика с плавающей точкой (sin, cos, tan, log, exp, sqrt, ...)

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

```
% ar -t /usr/lib/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



Использование статических библиотек

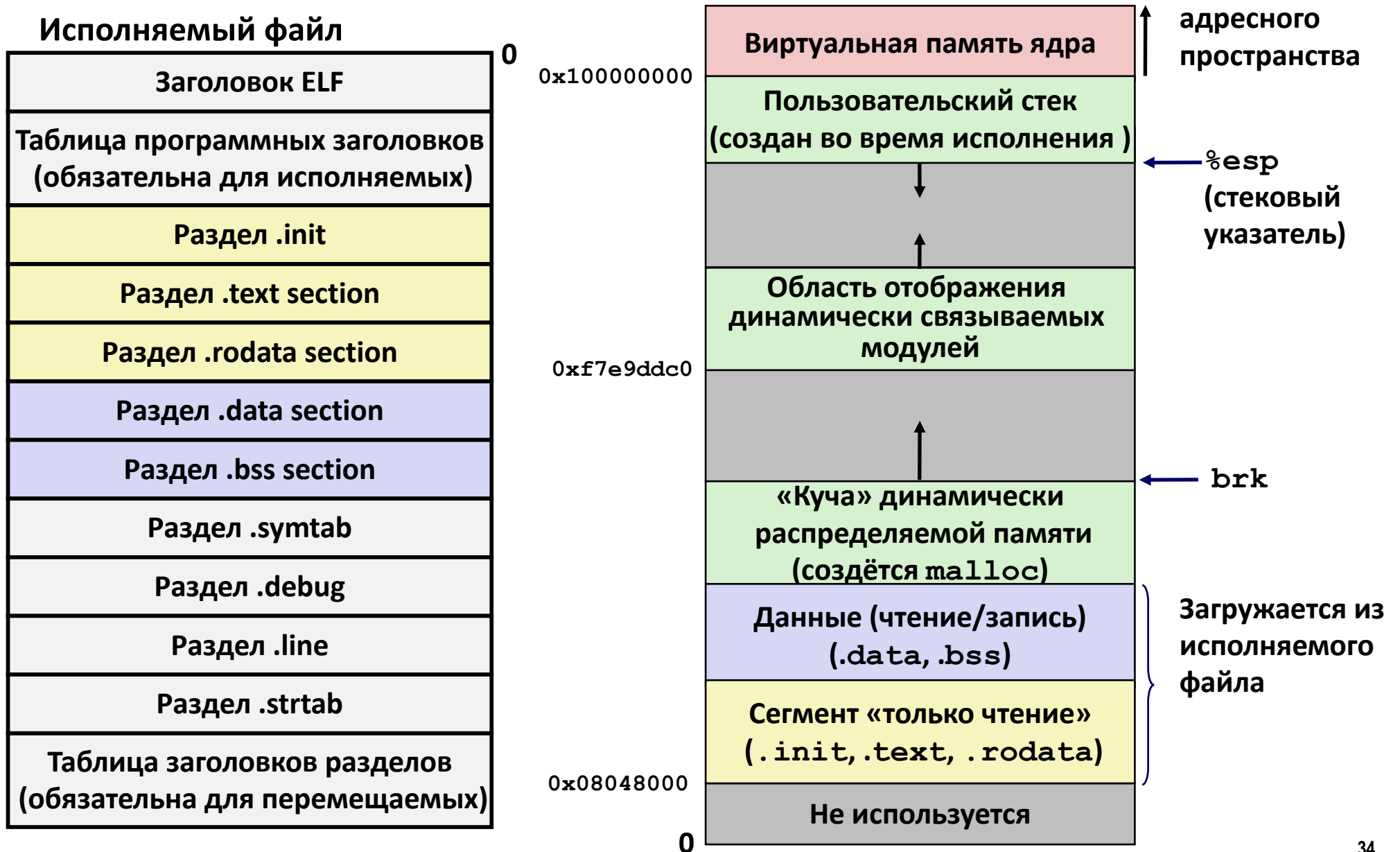
- **Алгоритм разрешения внешних ссылок редактором связей:**
 - Сканировать `.o` файлы и `.a` в порядке, указанном в командной строке.
 - Вести список неразрешённых ссылок.
 - В каждом новом `.o` или `.a` файле, пытаться разрешить каждую неразрешённую ссылку из списка символами определёнными в новом файле.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

- **Проблема:**

- Порядок указания в командной строке имеет значение!

- ```
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'
```

# Загрузка исполняемых файлов



# Разделяемые библиотеки – 1

## ■ Статические библиотеки имеют недостатки:

- Тиражирование хранимого кода (каждой функции необходимы стандартная libc)
- Тиражирование исполняемого кода
- Небольшие исправления в системных библиотеках требуют явной пересборки каждого приложения

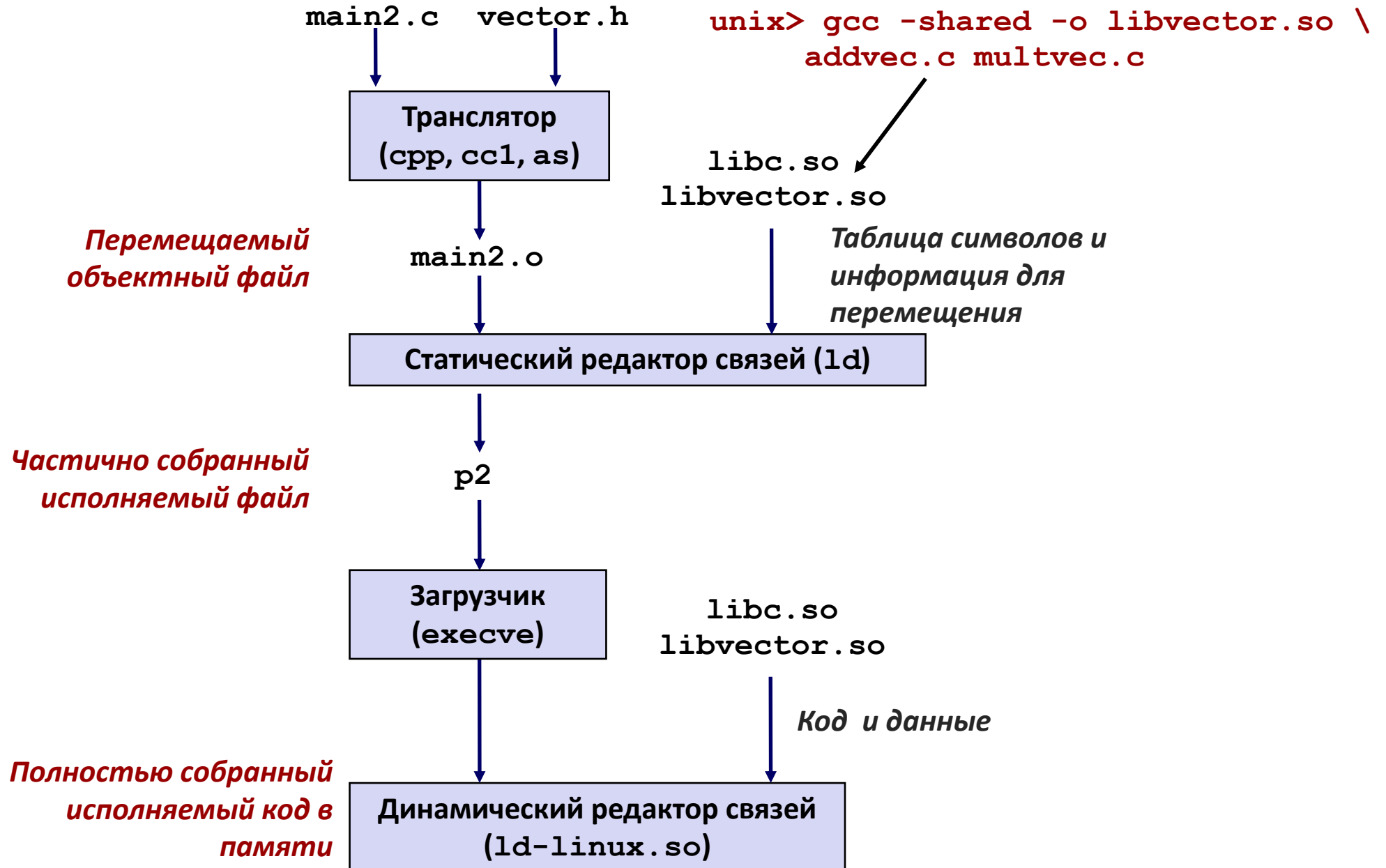
## ■ Решение: разделяемые библиотеки (Shared Libraries)

- Файлы содержащие код и данные, загружаемые и связываемые к приложением *динамически*, либо *во время загрузки*, либо *во время исполнения*
- Другие названия: динамически связываемые библиотеки, DLL, .so файлы

# Разделяемые библиотеки – 2

- **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()`
    - 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

```
#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;

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

# Динамическое связывание при исполнении

```
...

/* 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;
}
```

# Пример: «библиотечное посредничество»

- **Library interpositioning** - мощная техника связывания. Позволяет программисту перехватывать вызовы произвольных функций
- **Посредничество 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 посредничества

## ■ Безопасность

- Безопасный контейнер (песочница)
  - Опосредует обращения к стандартным функциям libc.
- Фоновое шифрование
  - Автоматическое шифрование изначально нешифрованных сетевых соединений.

## ■ Наблюдение и профилирование

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

# Пример программы

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

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

hello.c

- Цель: трассировка адресов и размеров выделяемых и освобождаемых блоков, без изменения исходного кода.
- Три решения: опосредование функций `malloc` и `free` во время компиляции, статического связывания и загрузки/исполнения.

# Посредничество во время компиляции

```
#ifdef COMPILETIME
/* Посредничество при компиляции с использованием
 * препроцессора. Местный malloc.h определяет обёртки для
 * malloc (free) как mymalloc (myfree) соответственно.
 */

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

/*
 * mymalloc - обёртка функции 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

# Посредничество 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
```

# Посредничество во время связывания

```
#ifdef LINKTIME
/* Посредничество во время связывания с использованием флага
 * статического редактора связей (ld) "--wrap symbol" */

#include <stdio.h>

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

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

mymalloc.c

# Посредничество во время связывания

```
linux> make hello1
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hello1 hello.c mymalloc.o
linux> make run1
./hello1
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- Флаг “-Wl” передаёт аргумент редактору связей
- “--wrap,malloc” требует разрешения ссылок особым образом :
  - Упоминание `malloc` должно разрешаться как `__wrap_malloc`
  - Упоминание `__real_malloc` должно разрешаться как `malloc`

```
#ifndef RUNTIME
/* Опосредование malloc и free на основе LD_PRELOAD
 * механизма динамического редактора связей (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;

 /* получение адреса libc malloc */
 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;
}
```

## Посредничество во время загрузки/исполнения

mymalloc.c

# Посредничество при загрузке/исполнении

```
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 среды `LD_PRELOAD` указывает динамическому редактору связей сначала разрешать упоминания (например для `malloc`) в `libdl.so` и `mymalloc.so`
  - `libdl.so` необходима для разрешения символов функций `dlopen`



# Посредничество: сводка

## ■ Во время компиляции

- Видимые вызовы malloc/free с помощью макрорасширений преобразуются в вызовы mymalloc/myfree

## ■ Во время 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