

Лекция 23

2 мая

Схема работы ассемблера

- Проблема: опережающие ссылки
- Первый проход: составление таблицы символов
 - Символ
 - Метка
 - Значение, которому приписано имя
 - Таблица символов
 - Длина поля
 - Глобальный/локальный
 - Перераспределение
- Второй проход: построение объектного кода

Пример Си-программы

main.c

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

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

swap.c

```
extern int buf[];

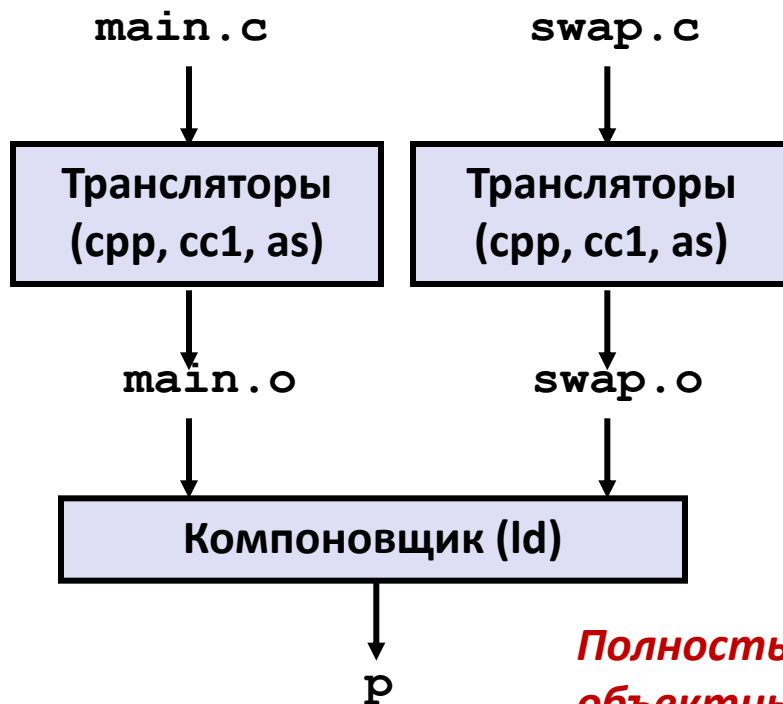
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

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

Статическая компоновка

- Программа транслируется и компоуется драйвером компилятора:
 - `unix> gcc -O2 -g -o p main.c swap.c`
 - `unix> ./p`



Файлы с исходным кодом

*Раздельно откомпилированные
файлы с перемещаемым
объектным кодом*

*Полностью скомпонованный исполняемый
объектный файл
(содержит код и данные всех функций
определенные в `main.c` и `swap.c`)*

Почему нужен компоновщик?

- Причина 1: Модульность программы
 - Программа может быть организована как набор небольших файлов с исходным кодом, а не один монолитный файл.
 - Есть возможность организовывать библиотеки функций, являющихся общими для разных программ
 - например, библиотека математических функций, стандартная библиотека языка Си

Почему нужен компоновщик?

- Причина 2: Эффективность
 - Время: Раздельная компиляция
 - Меняем код в одном файле, компилируем только его, повторяем компоновку
 - Нет необходимости повторять компиляцию остальных файлов с исходным кодом.
 - Место на диске: Библиотеки
 - Общие функции можно объединить в одном файле...
 - Исполняемые файлы и образ программы в памяти содержит только те функции, которые действительно используются.

Что делает компоновщик?

- Шаг 1. Разрешение символов

- В программе определяют и используют *символы* (переменные и функции):

- `void swap() {...} /* определение символа swap */`
- `swap(); /* ссылка на символ */`
- `int *xp = &x; /* определение символа xp, ссылка на x */`

- Определения символов сохраняются в таблице символов.

- Таблица символов – массив структур
- Каждая запись содержит имя, размер, позицию символа.

- Компоновщик устанавливает связь каждой ссылки на символ с единственным определением символа.

Что делает компоновщик?

- Шаг 2. Перемещение
 - Несколько объявлений секций кода и данных объединяются в единые секции
 - Символы перемещаются с их относительных позиций в .o-файлах на абсолютные адреса в исполняемом файле.
 - Обновляются все ссылки на символы, согласно их новым позициям.

Три типа объектных файлов (модулей)

- Перемещаемые объектные файлы (.o-файлы)
 - Содержит код и данные в форме, позволяющей проводить компоновку с другими перемещаемым объектными файлами.
 - Каждый .o-файл производится из **одного** файла с исходным кодом (.c-файла)
- Исполняемые объектные файлы (a.out-файлы)
 - Содержит код и данные в такой форме, что их можно напрямую копировать в память и запускать выполнение программы.
- Разделяемые объектные файлы (.so-файлы)
 - Особый вид перемещаемого объектного файла, который может быть загружен в память и скомпонован с программой динамически, во время ее загрузки и во время работы.
 - Windows - Dynamic Link Libraries (DLL)

Executable and Linkable Format (ELF)

- Стандартный бинарный формат объектных файлов
- Был предложен в AT&T System V Unix
 - Позже был поддержан в BSD и Linux
- Единый формат для
 - Перемещаемых объектных файлов (`.o`),
 - Исполняемых объектных файлов (`a.out`)
 - Разделяемых объектных файлов (`.so`)

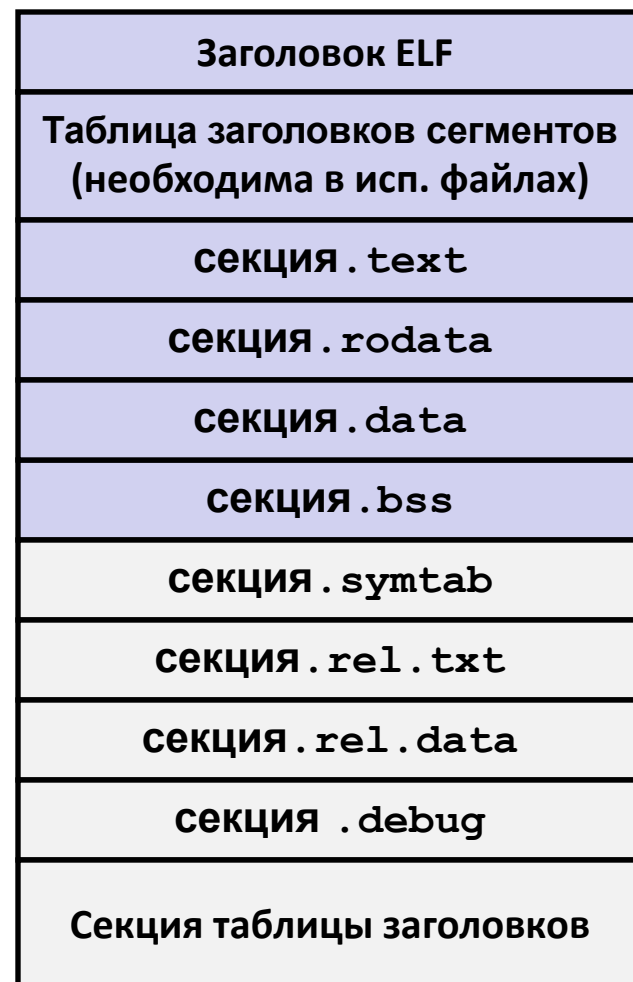
Формат ELF файла

- Заголовок Elf
 - Размер машинного слова, порядок байт, тип файла (.o, исп., .so), и др.
- Таблица заголовков сегментов
 - Размер страницы, сегменты виртуальной памяти, размеры сегментов.
- Секция .text
 - код
- Секция .rodata
 - Данные, доступные только на чтение: таблицы переходов, константы
- Секция .data
 - Инициализированные глобальные переменные
- Секция .bss
 - Неинициализированные глобальные переменные
 - У секции есть заголовок, на сама секция не занимает места

Заголовок ELF
Таблица заголовков сегментов (необходима в исп. файлах)
секция .text
секция .rodata
секция .data
секция .bss
секция .symtab
секция .rel.txt
секция .rel.data
секция .debug
Секция таблицы заголовков

Формат ELF файла (продолжение)

- Секция `.symtab`
 - Таблица символов
 - Имена функций и статических переменных
 - Имена секций
- Секция `.rel.text`
 - Данные для перемещения секции `.text`
 - Адреса инструкций которые должны быть обновлены
- Секция `.rel.data`
 - Данные для перемещения секции `.data`
 - Адреса глобальных переменных, инициализированных ссылками на внешние функции или глобальные переменные
- Секция `.debug`
 - Данные для символьного отладчика (`gcc -g`)
- Секция таблицы заголовков
 - Смещения и размеры каждой секции

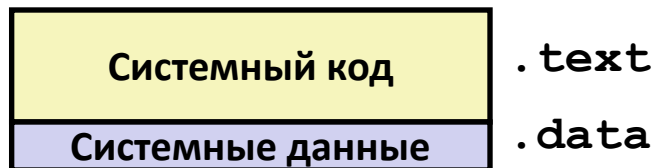


Символы в процессе компоновки

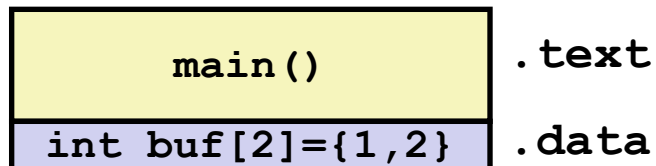
- Глобальные символы
 - Символы определенные в одном модуле таким образом, что их можно использовать в других модулях.
 - Например: не-**static** Си-функции и не-**static** глобальные переменные.
- Внешние символы
 - Глобальные символы, которые используются в модуле, но определены в каком-то другом модуле.
- Локальные символы
 - Символы определены и используются исключительно в одном модуле.
 - Например: Си-функции и переменные, определенные с модификатором **static**.
 - **Локальные символы не являются локальными переменными Си-программы**

Перемещение кода и данных

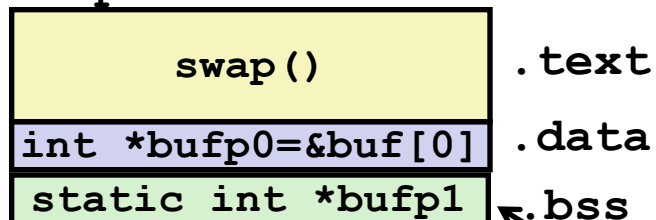
Перемещаемый объектный файл



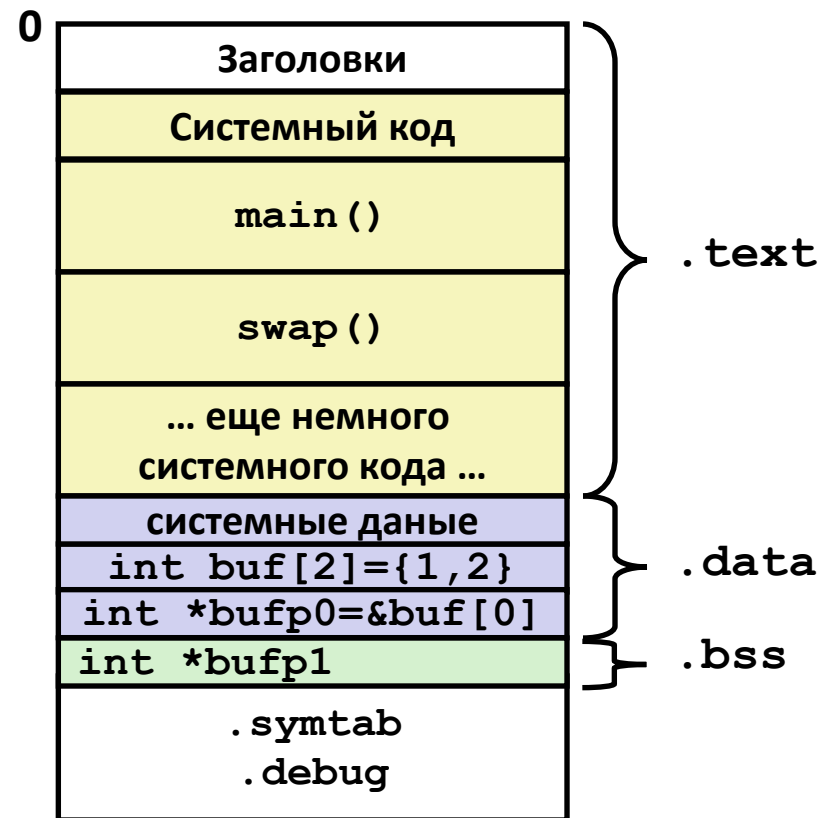
main.o



swap.o



Исполняемый объектный файл



Даже приватные данные файла swap, требуют размещения в .bss

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

Дизассемблирование секции .data:

objdump -r -d -M intel

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


Данные 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;
}
```

Дизассемблированная секция .data:

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

    0:  R_386_32 buf
```

Сильные и слабые символы

- Каждый символ в программе либо «сильный», либо «слабый»
 - **Сильные**: функции и инициализированные глобальные переменные
 - **Слабые**: неинициализированные глобальные переменные

сильный → `p1.c`
`int foo=5;`

сильный → `p1 () {`
`}`

`p2.c` ← **слабый**
`int foo;`

`p2 () {` ← **сильный**
`}`

Правила работы с символами

- Правило 1: Одинаковые сильные символы запрещены
 - Каждый элемент может быть определен только один раз
 - В противном случае ошибка компоновки
- Правило 2: Один сильный символ и несколько слабых – выбираем сильный символ
 - Ссылки на слабые символы заменяются ссылками на сильный символ
- Правило 3: Если несколько слабых символов, выбираем произвольный
 - Поведение можно поменять `gcc -fno-common`

Пример с заголовочным файлом

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;
}
```

Препроцессирование

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 выполнить подстановку файла

Роль заголовочных файлов (.h-файлов)

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 произойдет:

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

??

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

\

```
-DINITIALIZE
```

??

Глобальные переменные

- Следует избегать, если только есть такая возможность
- В противном случае
 - Используйте **static** если это возможно
 - Если определяете глобальную переменную, инициализируйте ее
 - Используйте **extern** если ссылаетесь на внешнюю глобальную переменную

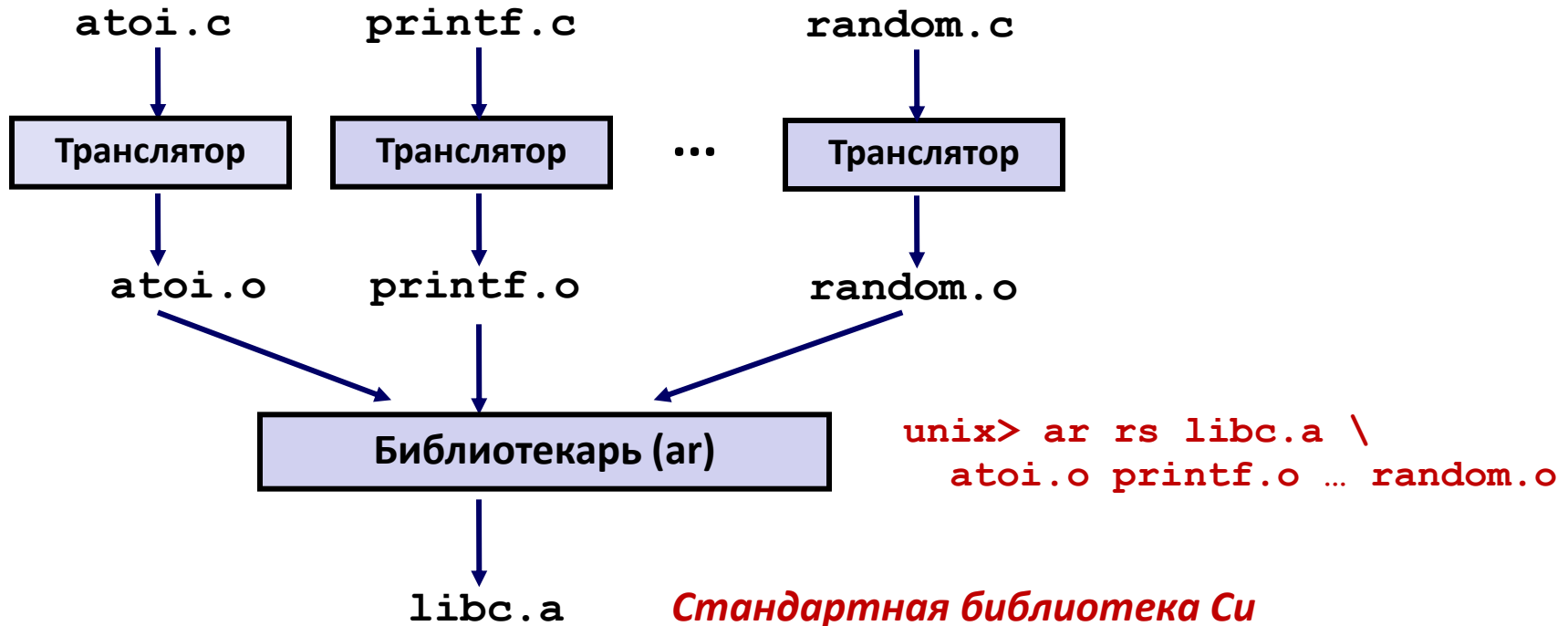
Работа с общими функциями

- Как следует размещать функции, часто используемые разными программами?
 - Математика, I/O, управление памятью, работа со строками, и т.д.
- Исходя из порядка компоновки:
 - **Вариант 1:** Поместить все функции в один файл
 - Компонуемся с одним большим объектным файлом
 - Неэффективно
 - **Вариант 2:** Поместить каждую функцию в отдельный файл
 - Во время компоновки явно указываем нужные объектные файлы
 - Более эффективно, но крайне неудобно для программиста

Решение: статические библиотеки

- **Статические библиотеки** (.a – файлы-архивы)
 - Близкие по смыслу перемещаемые объектные файлы группируются в одном файле, в т.н. называемом архиве.
 - Компоновщику указывают набор архивов для того, чтоб он попытался найти в них код с недостающими символами.
 - Если содержащийся в архиве файл помогает разрешить символ, то его автоматически включают в компоновку.

Создание статической библиотеки



- Библиотекарь позволяет выполнять инкрементальное обновление
- Повторная компиляция функции и замена соответствующего о-файла в библиотеке.

Часто используемые библиотеки

`libc.a` (Стандартная библиотека Си)

- 8 МБ архив из 1392 объектных файлов.
- I/O, управление памятью, работа со строками, даты и время, случайные числа, целочисленные математические функции

`libm.a` (Математическая библиотека Си)

- 1 МБ архив из 401 объектных файлов.
- Математические функции над числами с плавающей точкой (`sin`, `cos`, `tan`, `log`, `exp`, `sqrt`, ...)

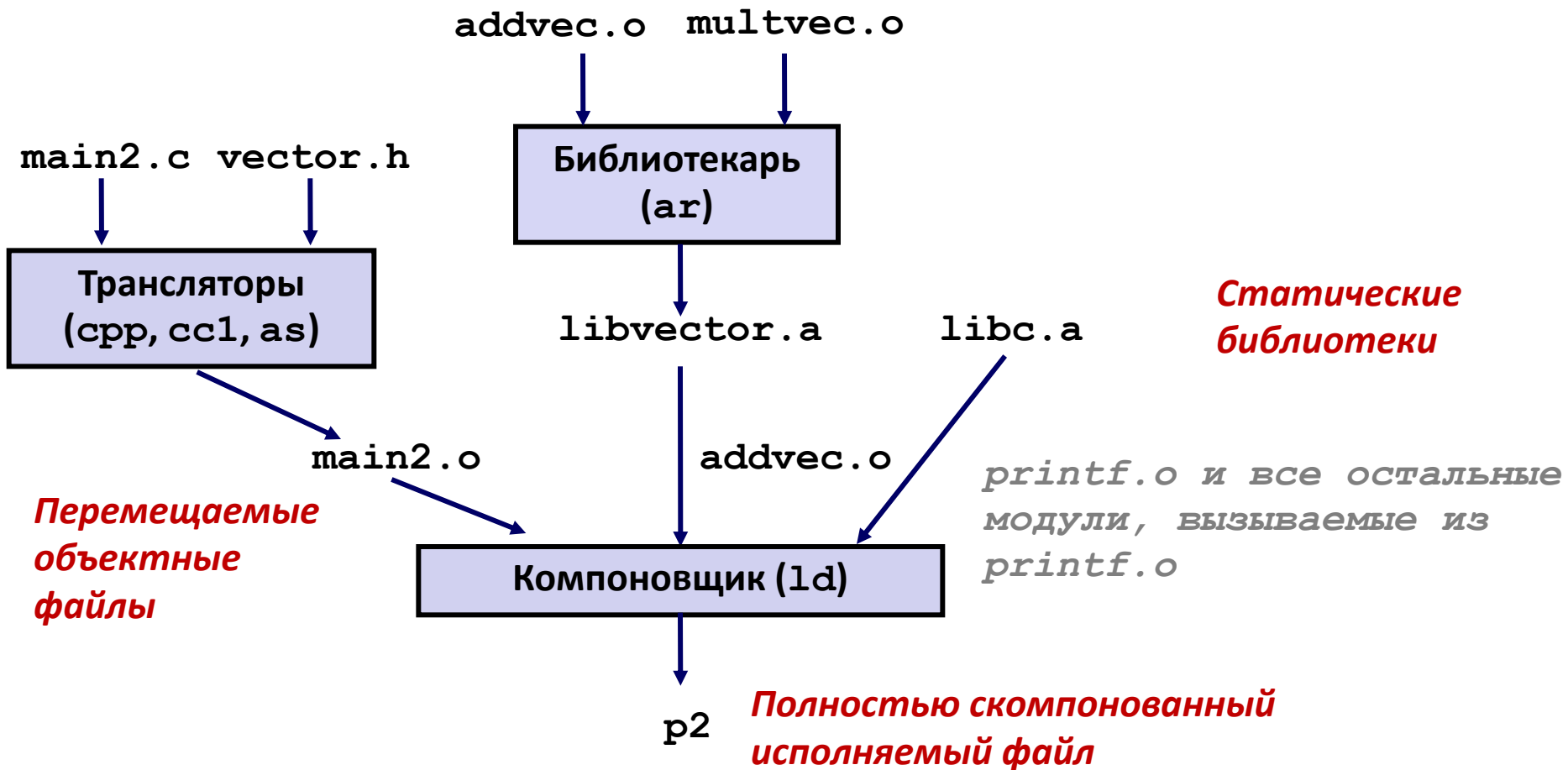
```
% ar -t /usr/lib/libc.a | sort
```

```
...  
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  
...
```

Компоновка со статическими библиотеками



Использование статических библиотек

- Алгоритм компоновщика для разрешения внешних ссылок :
 - Просматриваем `.o` файлы и `.a` файлы в порядке их следования в командной строке.
 - В процессе просмотра, поддерживаем список неразрешенных в данный момент символов.
 - Как только появляется новый `.o` или `.a` файл, пытаемся разрешить каждый еще неразрешенный символ среди символов, определенных в найденном файле.
 - Ошибка линковки, если по окончании просмотра остался хоть один неразрешенный символ.
- Проблема:
 - Важен порядок объектных файлов в командной строке!
 - Решение: помещать все библиотеки в конец 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'
```

Загрузка исполняемого объектного файла

Исполняемый объектный файл

0	ELF заголовок
	Таблица заголовков сегментов
	.init
	.text
	.rodata
	.data
	.bss
	.symtab
	.debug
	.line
	.strtab
	Секция таблицы заголовков

