Лекция 24

27 апреля

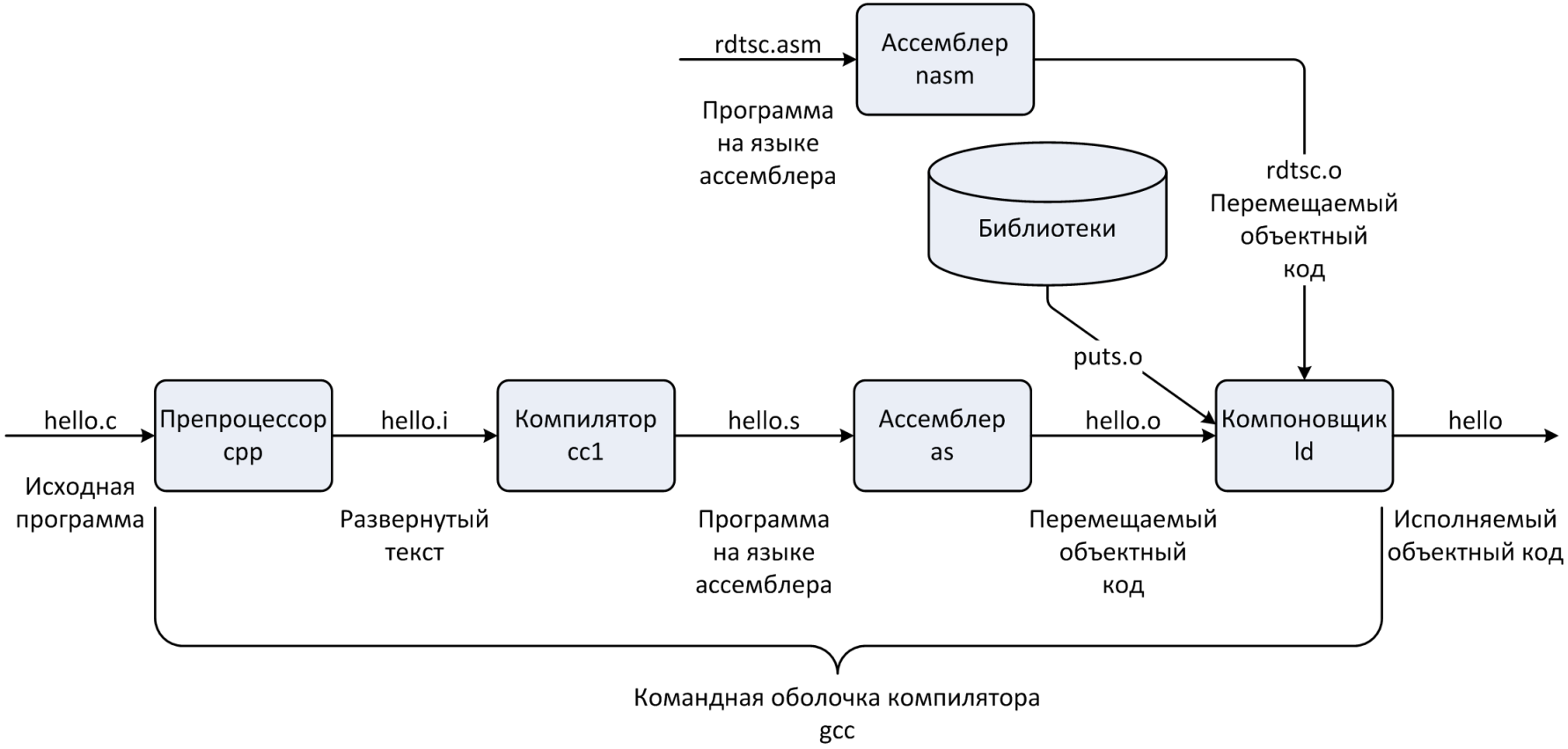
Язык программирования

- Цели, задачи, реализация языка
- Семантика
  - Операционная
- Компилируемые и интерпретируемые языки
  - Язык Java
    исходный код → байт-код → машинные команды
- Процессор – интерпретатор машинных команд
  - Эффективность написания программы в машинных командах крайне низкая
    - Кодирование команд
    - Организация программы
    - Организация вычислений

Система программирования

- Системные/прикладные программы
  - Операционная система
  - Программные средства разработки
- Система программирования – комплекс средств
  - Язык программирования
    - Информационные ресурсы
    - Программные инструменты
    - Библиотеки
- Этапы жизненного цикла программы
  - Проектирование
    - Сбор и анализ требований к программе
    - Разработка
  - Реализация
    - Кодирование
    - Отладка
  - Сопровождение

Система программирования языка Си



Выполнение программы

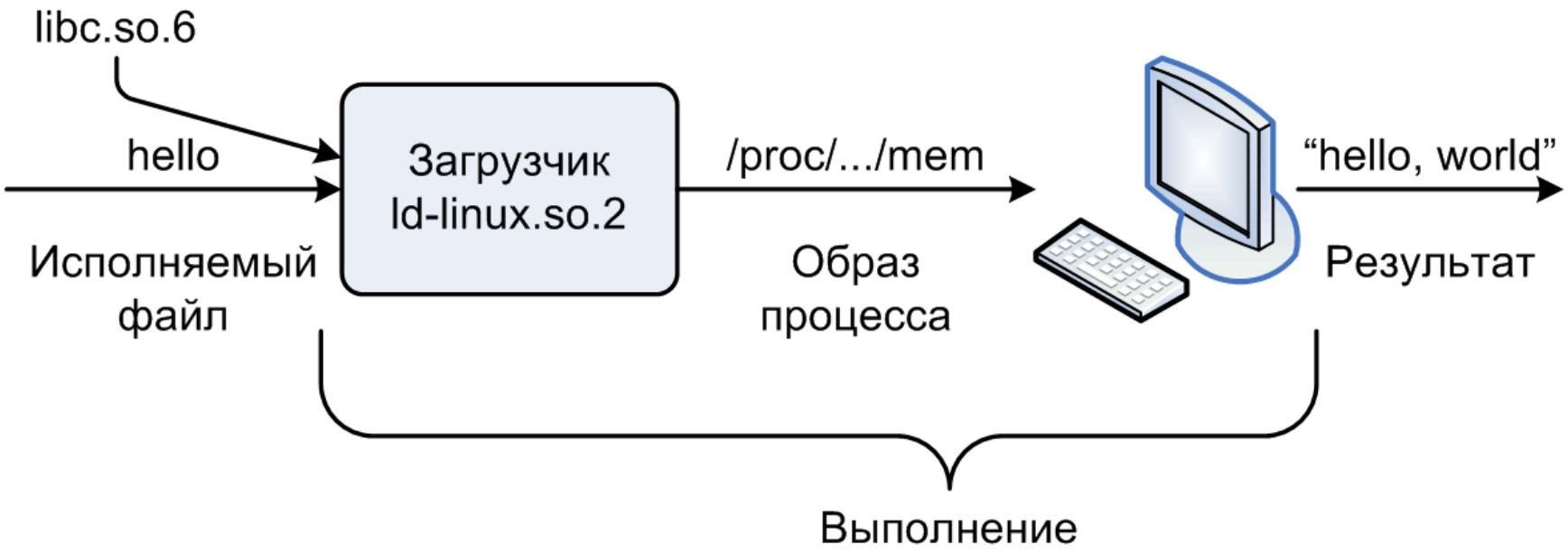


Схема работы ассемблера

- 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

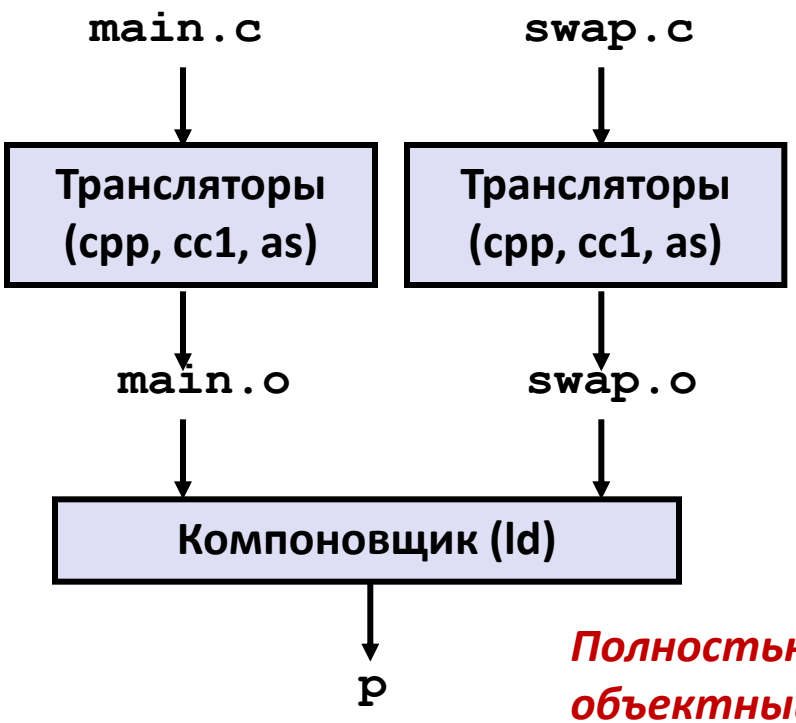
```
int buf[2] = {1, 2};

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 компонуется 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 (**.** ○-файлы)
  - Содержит код и данные в форме, позволяющей проводить компоновку с другими перемещаемым объектными файлами.
    - Каждый **.** ○-файл производится из **одного** файла с исходным кодом (**.** ⊂-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.

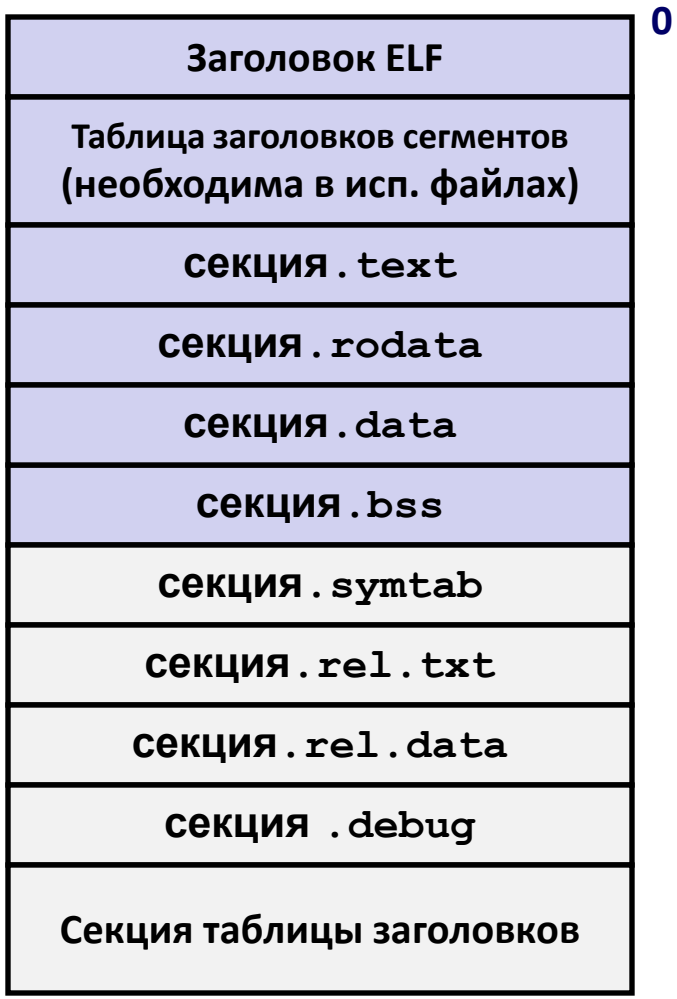
– Разделяемые объектные файлы (.so-файлы)
  - Особый вид перемещаемого объектного файла, который может быть загружен в память и скомпонован с программой динамически, во время ее загрузки и во время работы.
  - Windows - Dynamic Link Libraries (DLL)

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`)

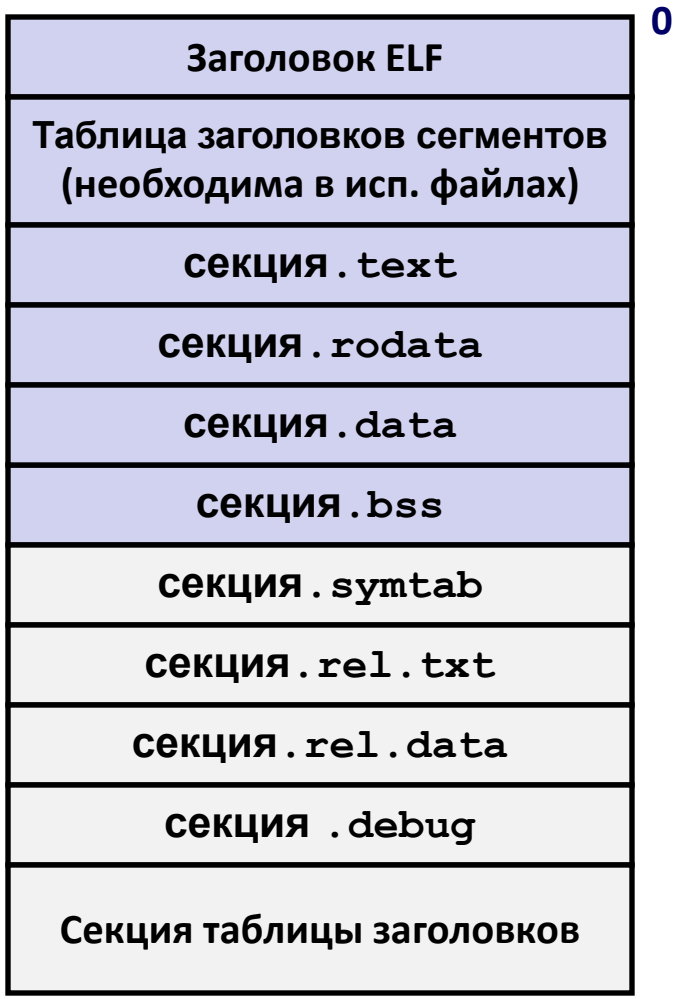
Формат ELF файла

- 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



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**

Разрешение символов

Глобальные

Глобальный

Внешний

Локальный

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

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

main.c

Внешний

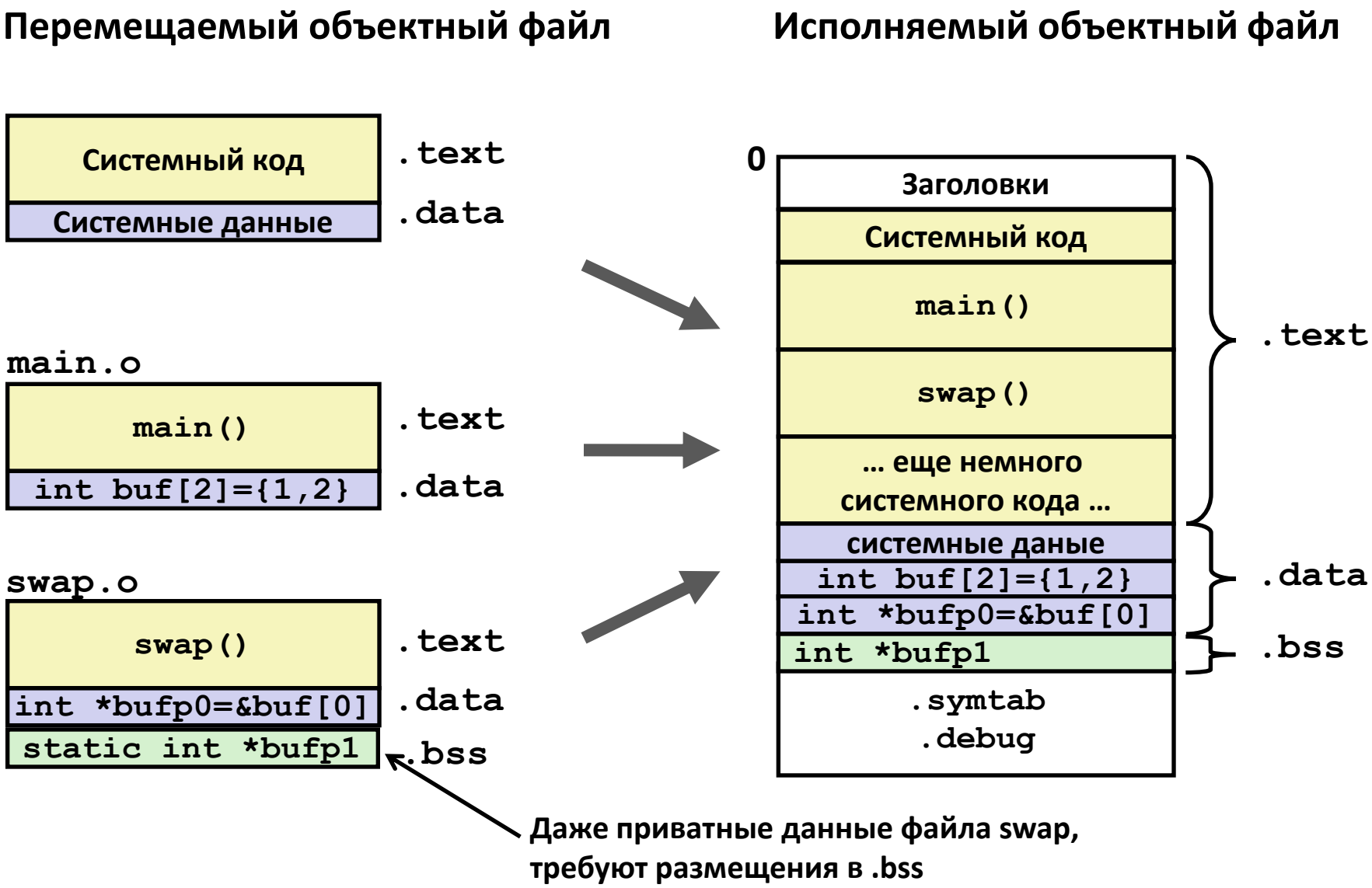
```
extern int buf[];

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

void swap()
{
  int temp;

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

Глобальный

Для компоновщика переменной temp не сущществует

swap.c

Перемещение кода и данных



Данные для перемещения (main)

main.c

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

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

main.o

```
0000000 <main>:
   0:   8d 4c 24 04        lea    ecx, [0x4 + esp]
   4:   83 e4 f0           and    esp, 0xfffffff0
   7:   ff 71 fc           push   dword [0xfffffffc + 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, [0xfffffffc + ecx]
  20:   c3                 ret
```

```
objdump -r -d -M intel
```

```
Дизассемблирование секции .data:

 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

p1.c

сильный →
```
int foo=5;
```
сильный →
```
p1() {
}
```

p2.c

```
int foo;
```
← **слабый**

```
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

Вопросы к залу

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

```
p1() {}
```

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

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

```
int x;
int y;
p1() {}
```

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

```
int x=7;
int y=5;
p1() {}
```

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

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

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

Вопросы к залу

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

```
p1() {}
```

Ошибка компоновки: два сильных символа (**p1**)

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

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

Ссылки на **x** будут ссылаться на один и тот же неинициализированный int. Но какой?

```
int x;
int y;
p1() {}
```

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

Запись в **x** (**p2**) может поменять **y**!

```
int x=7;
int y=5;
p1() {}
```

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

Запись в **x** (**p2**) обязательно поменяет **y**!

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

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

Ссылка на **x** будет ссылаться на один и тот же инициализированный int.

Наихудший сценарий: два одинаковые «слабые» структуры, Откомпилированные разными компиляторами с разными правилами выравнивания.

Пример с заголовочным файлом

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
```

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

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

Без инициализации

```
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

```
#ifdef 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

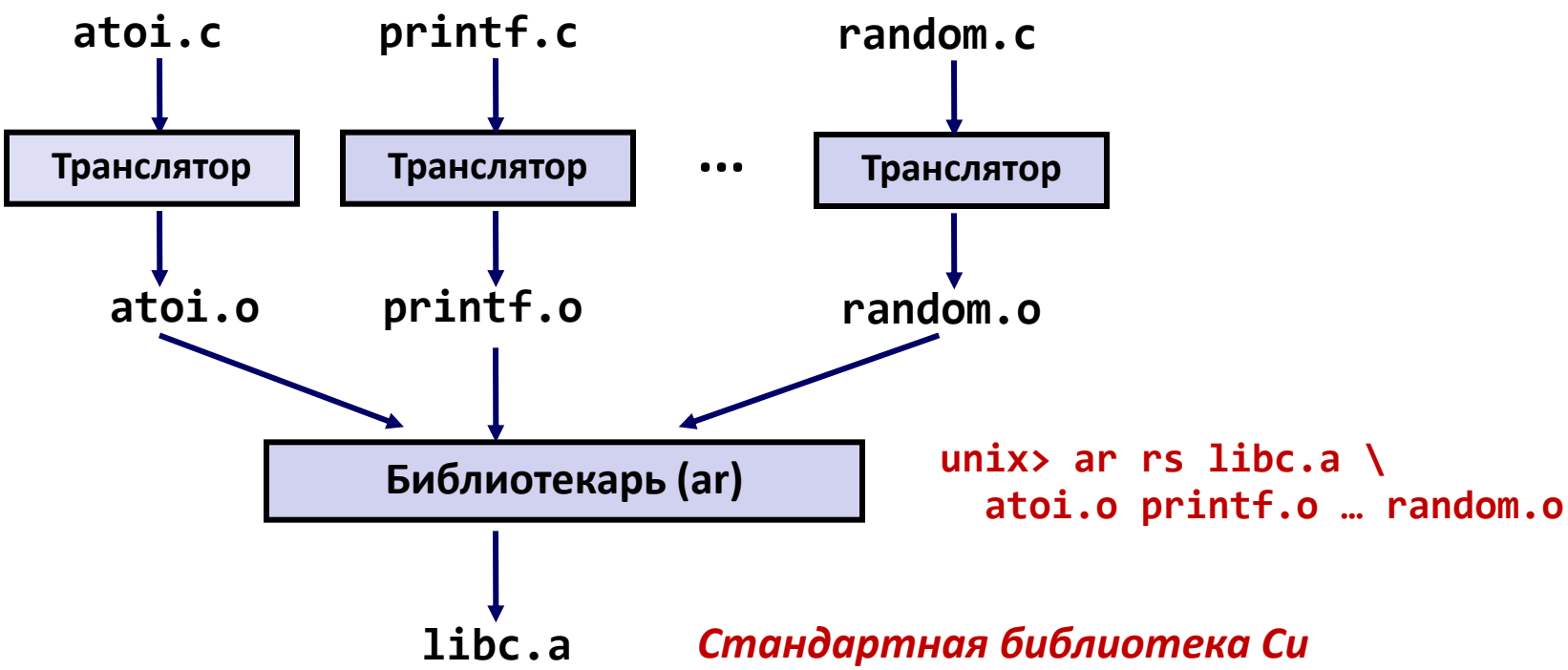
- Как следует размещать функции, часто используемые разными программами?
  - Математика, I/O, управление памятью, работа со строками, и т.д.

- 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

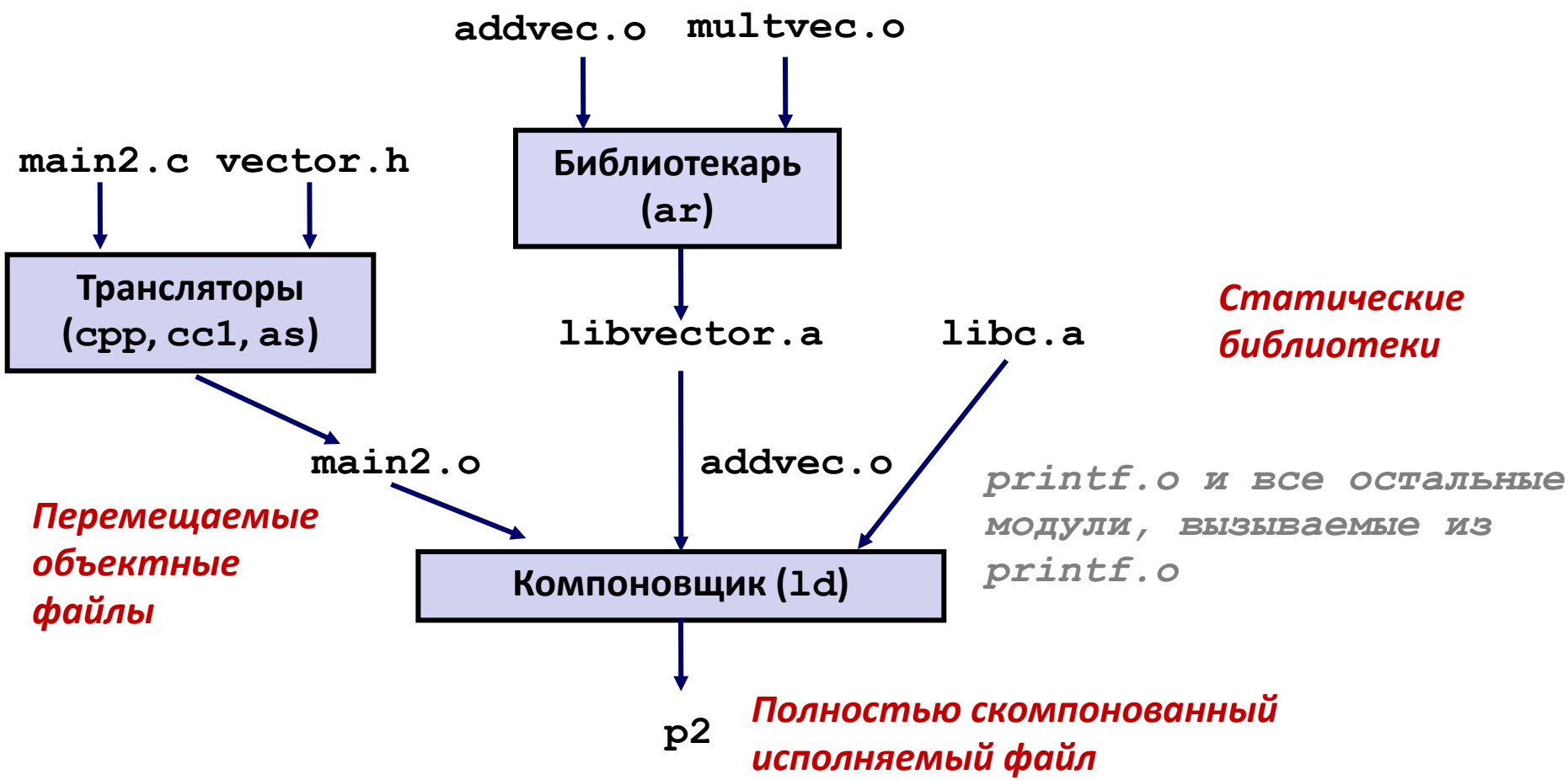
`libm.a` (Математическая библиотека Си)

- 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