

Лекция 0x13

20 апреля

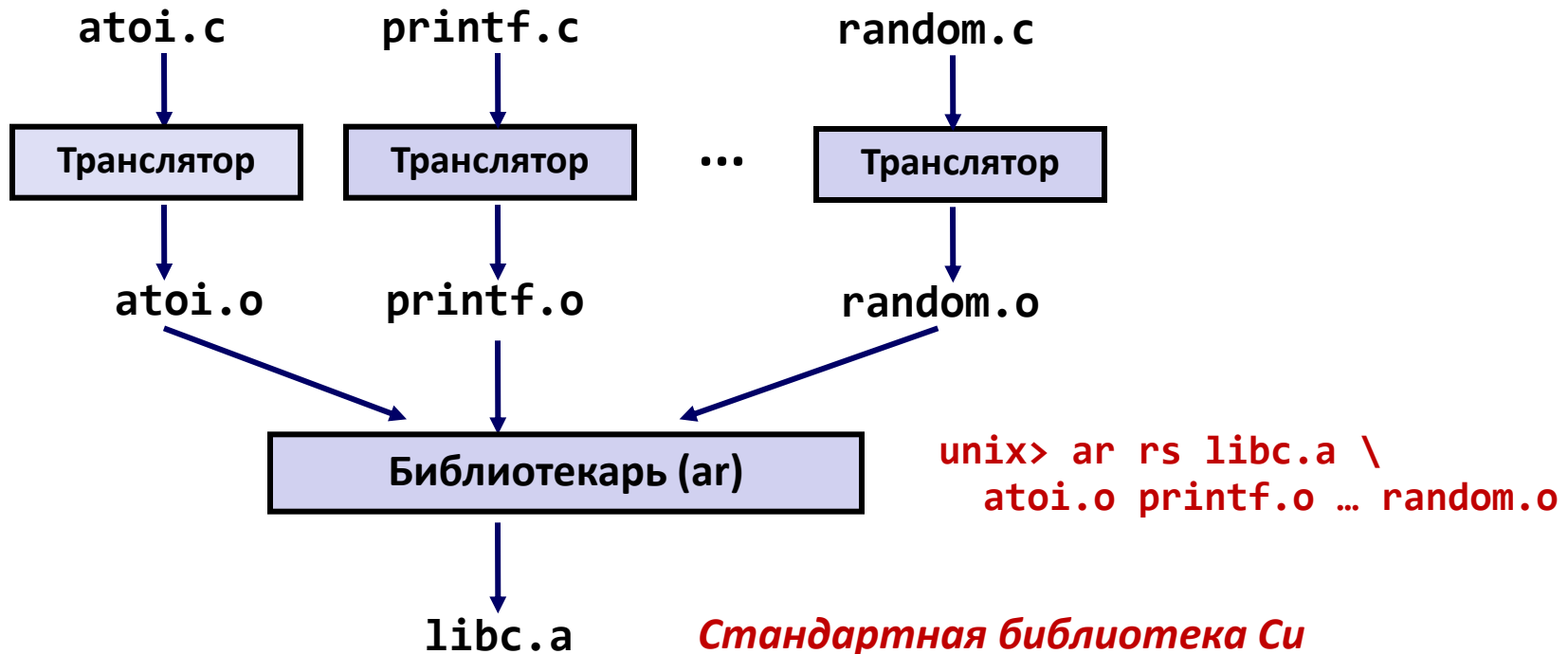
Работа с общими функциями

- 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

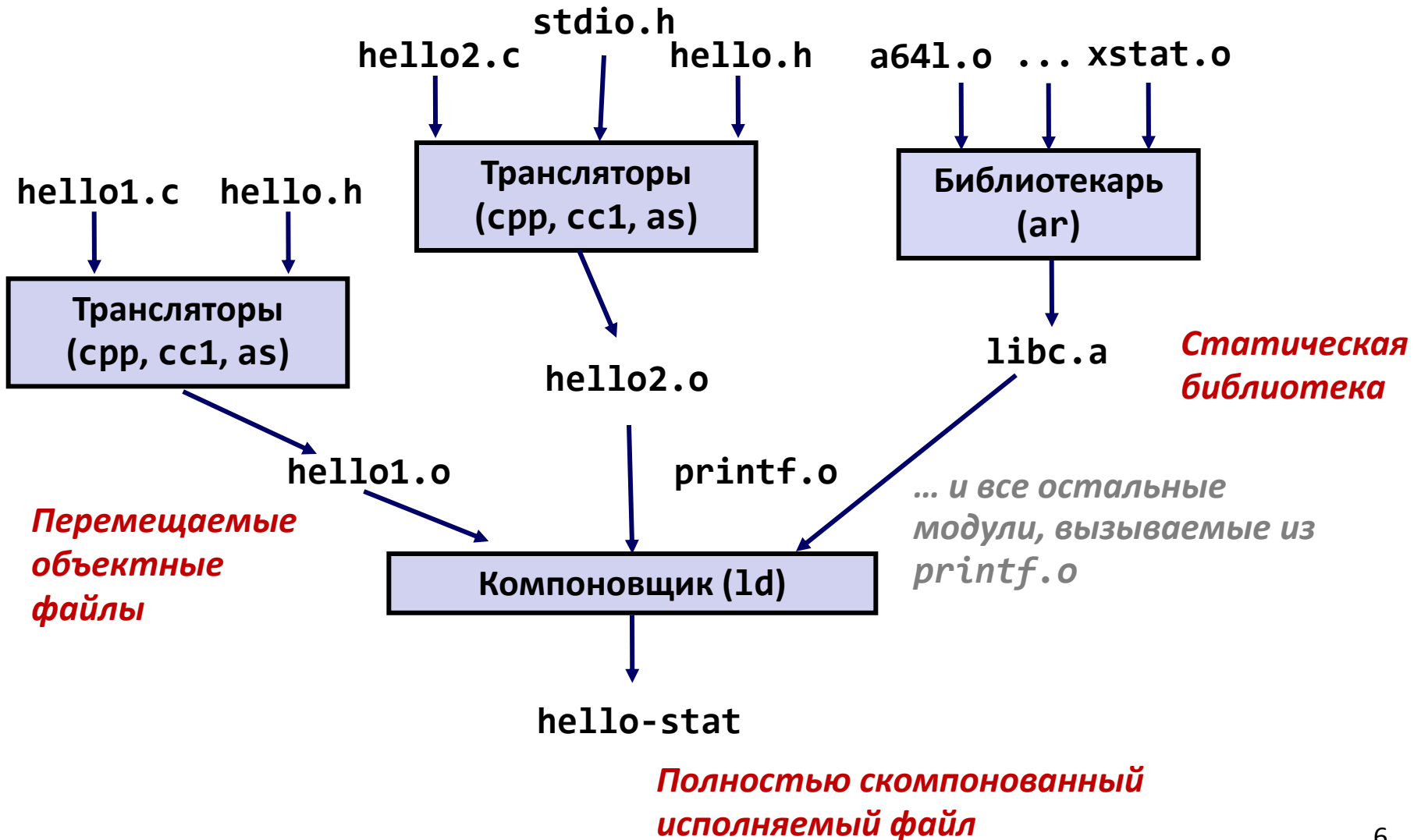
libm.a (Математическая библиотека Си)

- 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

- Подготовка адресного пространства для запускаемой программы
- Просмотр у заданного файла таблицы заголовков сегментов
 - Для каждого сегмента типа LOAD загружаем указанные байты из файла на указанные адреса памяти
- Передаем управление на точку входа в программу (символ `_start`)

```
typedef struct {
    Elf32_Word p_type;      /* LOAD, DYNAMIC, INTERP, PHDR, NULL, ... */
    Elf32_Off  p_offset;    /* Смещение от начала файла */
    Elf32_Addr p_vaddr;     /* Начальный (базовый) адрес в памяти */
    Elf32_Addr p_paddr;     /* Не используется */
    Elf32_Word p_filesz;    /* Размер в файле в байтах */
    Elf32_Word p_memsz;     /* Размер в памяти в байтах */
    Elf32_Word p_flags;     /* R/W/X */
    Elf32_Word p_align;     /* Величина выравнивания */
} Elf32_Phdr;
```


Связь секций и сегментов

```
snoop@jezek:~/samples/2017/linking$ readelf -l hello-static
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x8048a8d
```

```
There are 6 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x08048000	0x08048000	0xa3a31	0xa3a31	R E	0x1000
LOAD	0x0a3f3c	0x080ecf3c	0x080ecf3c	0x01124	0x02608	RW	0x1000
NOTE	0x0000f4	0x080480f4	0x080480f4	0x00044	0x00044	R	0x4
TLS	0x0a3f3c	0x080ecf3c	0x080ecf3c	0x00010	0x00028	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x0a3f3c	0x080ecf3c	0x080ecf3c	0x000c4	0x000c4	R	0x1

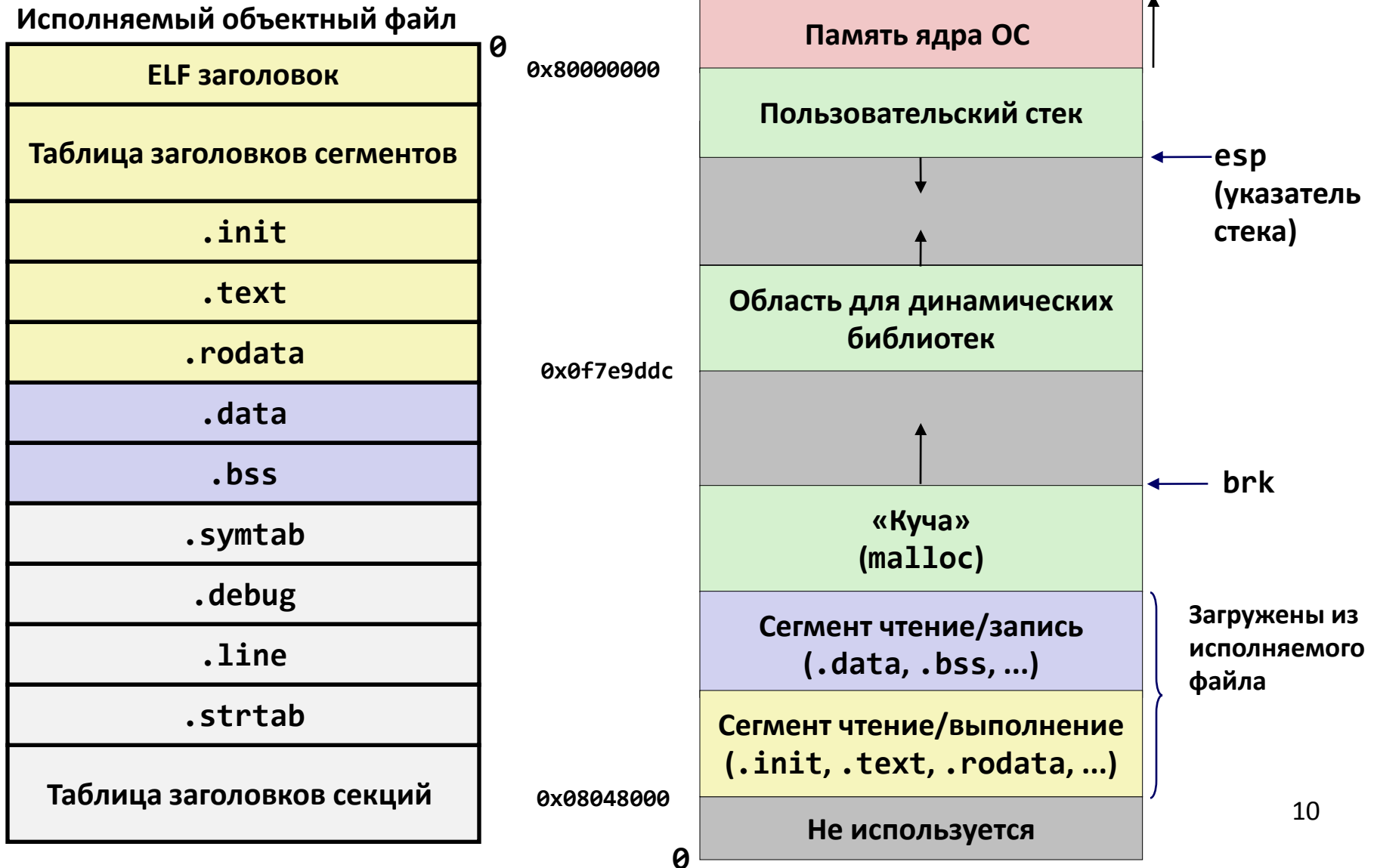
```
Section to Segment mapping:
```

```
Segment Sections...
```

```
00 ... .init .text .fini .rodata ...
```

```
01 ... .init_array .fini_array .data .bss
```

Загрузка исполняемого объектного файла



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

```
snoop@earth:~/samples$ gcc -static -o hello-static hello1.o hello2.o
snoop@jezek:~/samples$ ls -l hello-static
-rwxrwxr-x 1 snoop snoop 743476 Apr 12 09:15 hello-static
snoop@earth:~/samples$ gcc -o hello hello1.o hello2.o
snoop@jezek:~/samples$ ls -l hello
-rwxrwxr-x 1 snoop snoop 7404 Apr 12 09:06 hello
```

- Способ преодолеть эти недостатки: динамические библиотеки (shared libraries)
 - Объектные файлы, в которых содержатся код и данные компонуется 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 (динамическая загрузка библиотек).

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

- Функции динамических библиотек могут одновременно использоваться несколькими процессами.

Проблемы

- Динамическая библиотека может быть размещена в произвольном месте памяти
- Как обращаться из основной программы к переменным и функциям, размещение которых в памяти будет известно только в момент запуска программы?
 - До момента компоновки неизвестно, куда ведет ссылка – в перемещаемый код другой единицы трансляции или в динамическую библиотеку
 - Перемещаемый код с ссылками уже построен, в нем могут меняться только значения операндов (перебазируемые ссылки)
 - В случае динамической загрузки, адреса размещения будут определены еще позже – в уже работающей программе
- Как строить код динамической библиотеки, когда до момента начала выполнения ее кода неизвестно, по каким адресам будут размещены ее собственные функции и данные

```
#include "hello.h" hello1.c

char *buf = "Hello, world!\n";

int main() {
    int ret_code = 0;
    func();
    return ret_code;
}
```

```
#include <stdio.h> hello2.c
#include "hello.h"

void func() {
    printf("%s", buf);
}
```

```
extern void func(); hello.h

extern char* buf;
```

```
all: hello-dlib Makefile

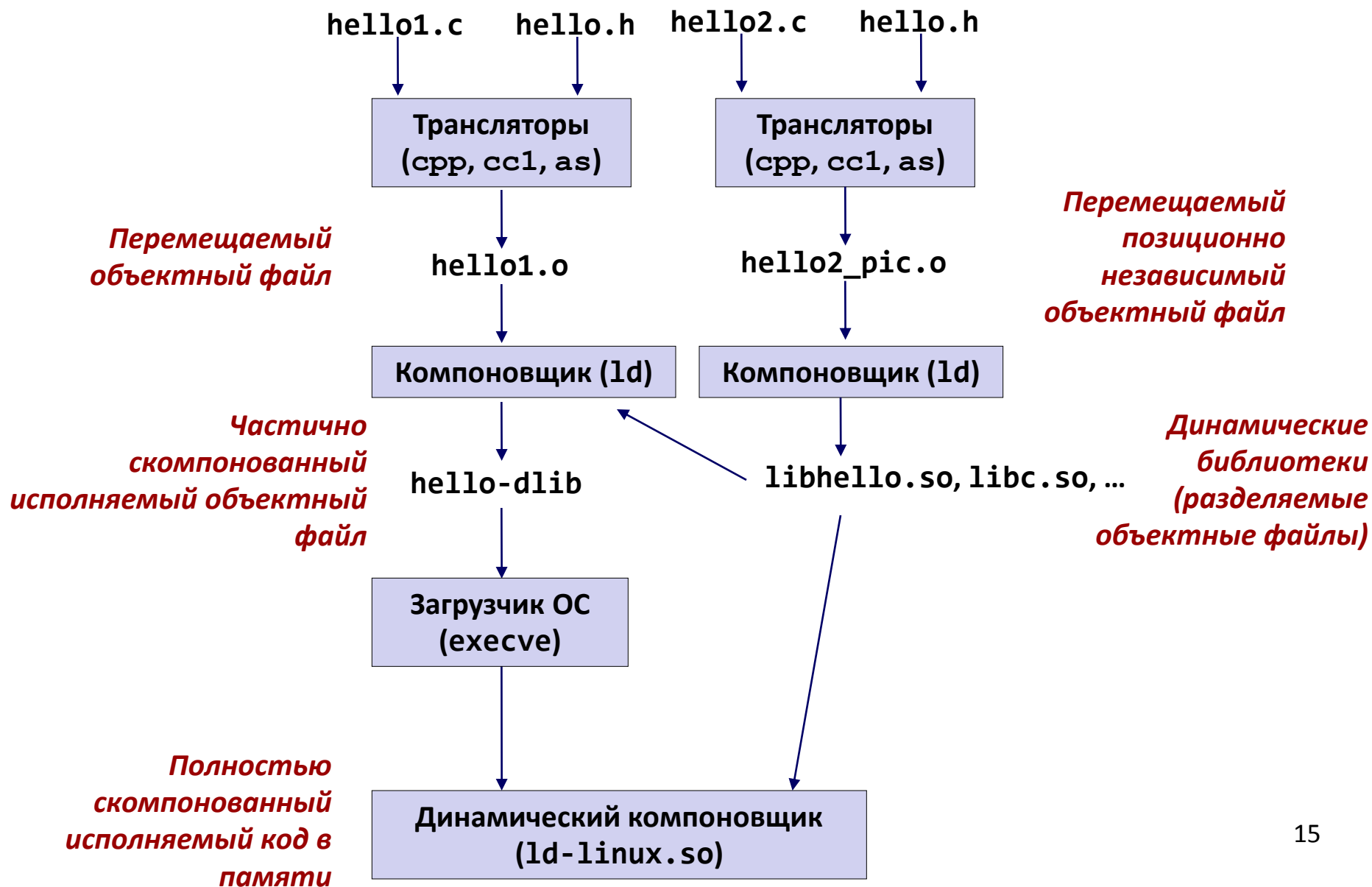
hello-dlib: hello1.o libhello.so
    gcc -o hello-dlib hello1.o libhello.so

libhello.so: hello2_pic.o
    gcc -shared -o libhello.so hello2_pic.o

hello2_pic.o: hello.h hello2.c
    gcc -c -fPIC -O3 -D_FORTIFY_SOURCE=0 -o hello2_pic.o hello2.c

hello1.o: hello.h hello1.c
    gcc -c -O3 -D_FORTIFY_SOURCE=0 -o hello1.o hello1.c
```

Динамическая компоновка времени загрузки



Позиционно независимый код в IA-32

- Абсолютные адреса неизвестны, но известны смещения.
 - Можно обращаться к функциям своего модуля.
 - В IA-32 нет возможности обратиться к данным по смещению относительно текущего значения счетчика команд. Такая возможность появится в x86_64.
- Код и данные размещаются в памяти друг за другом, в сегмент данных входит служебная таблица – Global Offset Table (GOT).
- GOT содержит указатели (абсолютные адреса) на внешние функции и переменные. Их адреса становятся известны не ранее этапа динамической компоновки при запуске программы.
- Динамический компоновщик заполняет GOT необходимыми значениями.



Как заставить ассемблер построить ПОЗИЦИОННО НЕЗАВИСИМЫЙ КОД?

```
extern void func();
```

```
hello.h
```

```
extern char* buf;
```

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

```
hello2.c
```

```
#include "hello.h"
```

```
void func() {
    printf("%s", buf);
}
```

Синтаксис ассемблера gas оставлен без изменений.

```
func:
```

```
    push    ebx
    call    __x86.get_pc_thunk.bx
    add     ebx, OFFSET FLAT: _GLOBAL_OFFSET_TABLE_
    sub     esp, 16
    mov     eax, DWORD PTR buf@GOT[ebx]
    push   DWORD PTR [eax]
    lea    eax, .LC0@GOTOFF[ebx]
    push   eax
    call   printf@PLT
    add     esp, 24
    pop    ebx
    ret
```

Определенные ключевые слова указывают ассемблеру на необходимость создания ссылок типа *R_386_GOTPC*, *R_386_GOT32*, *R_386_GOTOFF*, *R_386_PLT*

```
...
hello2_pic.s: hello.h hello2.c
    gcc -S -masm=intel -fPIC -O3 -D_FORTIFY_SOURCE=0 -o hello2_pic.s hello2.c

hello2_pic.o: hello2_pic.s
    gcc -c -o hello2_pic.o hello2_pic.s
...
```

Makefile был модифицирован для получения ассемблерного кода

```
snoop@jezek:~/samples/2017/linking$ objdump -d -r -M intel hello2_pic.o
hello2_pic.o:          file format elf32-i386
```

Как обратиться к данным по известному смещению относительно счетчика команд, если это не x86_64?

```
Disassembly of section .text:
```

```
00000000 <func>:
```

```

0:  53                push   ebx
1:  e8 fc ff ff ff   call  2 <func+0x2>
                2:  R_386_PC32    __x86.get_pc_thunk.bx
6:  81 c3 02 00 00 00 add   ebx,0x2
                8:  R_386_GOTPC   _GLOBAL_OFFSET_TABLE_
c:  83 ec 10         sub   esp,0x10
f:  8b 83 00 00 00 00 mov   eax,DWORD PTR [ebx+0x0]
                11: R_386_GOT32   buf
15: ff 30           push  DWORD PTR [eax]
17: 8d 83 00 00 00 00 lea  eax,[ebx+0x0]
                19: R_386_GOTOFF  .LC0
1d: 50             push  eax
1e: e8 fc ff ff ff   call  1f <func+0x1f>
                1f: R_386_PLT32   printf
23: 83 c4 18         add   esp,0x18
26: 5b             pop   ebx
27: c3             ret
```

```
snoop@jezek:~/samples/2017/linking$ objdump -d -r -M intel hello2_pic.o
hello2_pic.o:          file format elf32-i386
```

Disassembly of section .text:

00000000 <func>:

```

0:   53                push   ebx
1:   e8 fc ff ff ff    call   2 <func+0x2>
2:   R_386_PC32      __x86.get_pc_thunk.bx
6:   81 c3 02 00 00 00  add   ebx,0x2
8:   R_386_GOTPC    _GLOBAL_OFFSET_TABLE_
c:   83 ec 10
f:   8b 83 00 00 00 00
15:  ff 30
17:  8d 83 00 00 00 00
1d:  50
1e:  e8 fc ff ff
23:  83 c4 18
26:  5b
27:  c3                ret

```

- Тип перебазирувания R_386_GOTPC
новое значение ссылки = GOT + A - P
- GOT – абсолютный адрес памяти, связанный с некоторым элементом глобальной таблицы смещений
- A – дополнительное слагаемое (*addend*), хранимое непосредственно в байтах ссылки
- P – абсолютный адрес ссылки
- Наличие такого типа перебазирувания в файле требует от компоновщика создавать GOT
- Новое значение ссылки – расстояние до GOT. После сложения EBX будет указывать на некоторый элемент GOT.


```
snoop@jezek:~/samples/2017/linking$ objdump -d -r -M intel hello2_pic.o
```

```
hello2_pic.o:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000000 <func>:
```

```

0:  53                               push   ebx
1:  e8 fc ff ff ff                  call   1f <func+0x1f>
6:  81 c3 02 00 00 00               add   ecx,0x2
c:  83 ec 10                         cmp   ecx,0x10
f:  8b 83 00 00 00 00               mov   ebx,ebx
15: ff 30                            push  DWORD PTR [eax]
17: 8d 83 00 00 00 00               lea  eax,[ebx+0x0]
19: R_386_GOTOFF .LC0
1d: 50                               push  eax
1e: e8 fc ff ff ff                  call  1f <func+0x1f>
1f: R_386_PLT32 printf
23: 83 c4 18                         add   esp,0x18
26: 5b                               pop   ebx
27: c3                               ret

```

- Тип перебазирования R_386_GOTOFF
новое значение ссылки = $S + A - GOT$
- S – абсолютный адрес памяти, которому символ соответствует после перемещения
- GOT – абсолютный адрес памяти, связанный с некоторым элементом глобальной таблицы смещений
- A – дополнительное слагаемое (addend), хранимое непосредственно в байтах ссылки
- Наличие такого типа перебазирования в файле требует от компоновщика создавать GOT
- Новое значение ссылки – смещение от базового адреса `_GLOBAL_OFFSET_TABLE_` до адреса символа

```
noop@jezek:~/samples/2017/linking$ objdump -d -M intel libhello.so
...
00000570 <func>:
570:    53                push   ebx
571:    e8 ba fe ff ff   call  430 <__x86.get_pc_thunk.bx>
576:    81 c3 8a 1a 00 00 add   ebx,0x1a8a
57c:    83 ec 10         sub   esp,0x10
57f:    8b 83 f4 ff ff ff mov   eax,DWORD PTR [ebx-0xc]
585:    ff 30          push  DWORD PTR [eax]
587:    8d 83 ac e5 ff ff lea   eax,[ebx-0x1a54]
58d:    50                push  eax
58e:    e8 6d fe ff ff   call  400 <printf@plt>
593:    83 c4 18         add   esp,0x18
596:    5b                pop   ebx
597:    c3                ret
```

- Компоновщик создает GOT
- Все ссылки в коде обновляются
- Создается секция `.dynsym`, в которой собраны описания символов, используемых в динамической компоновке
- Создается секция `.rel.dyn`, в которой собраны описания новых ссылок, размещенных в GOT

```
snoop@jezek:~/samples/2017/linking$ readelf -s libhello.so
```

```
Symbol table '.dynsym' contains 14 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
5:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	buf
...							
13:	00000570	40	FUNC	GLOBAL	DEFAULT	11	func
...							

- Тип перебазирувания R_386_GLOB_DAT
НОВОЕ ЗНАЧЕНИЕ ССЫЛКИ = S
- S – абсолютный адрес памяти, связанный с СИМВОЛОМ

```
snoop@jezek:~/samples/2017/linking$ readelf -r libhello.so
```

```
Relocation section '.rel.dyn' at offset 0x360 contains 9 entries:
```

Offset	Info	Type	Sym.Value	Sym. Name
...				
00001ff4	00000506	R_386_GLOB_DAT	00000000	buf
...				

```
snoop@jezek:~/samples/2017/linking$ objdump -r -s -j .got libhello.so
```

```
libhello.so: file format elf32-i386
```

```
Contents of section .got:
```

```
1fe8 00000000 00000000 00000000 00000000 .....
1ff8 00000000 00000000 .....
.....
```

```
_GLOBAL_OFFSET_TABLE_ - 0xc = 0x2000 - 0xc = 0x1ff4
```

```
snoop@jezek:~/samples/2017/linking$ readelf -l libhello.so
```

```
Elf file type is DYN (Shared object file)
```

```
Entry point 0x430
```

```
There are 7 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00000000	0x00000000	0x00634	0x00634	R E	0x1000
LOAD	0x000efc	0x00001efc	0x00001efc	0x00120	0x00124	RW	0x1000

```
...
```

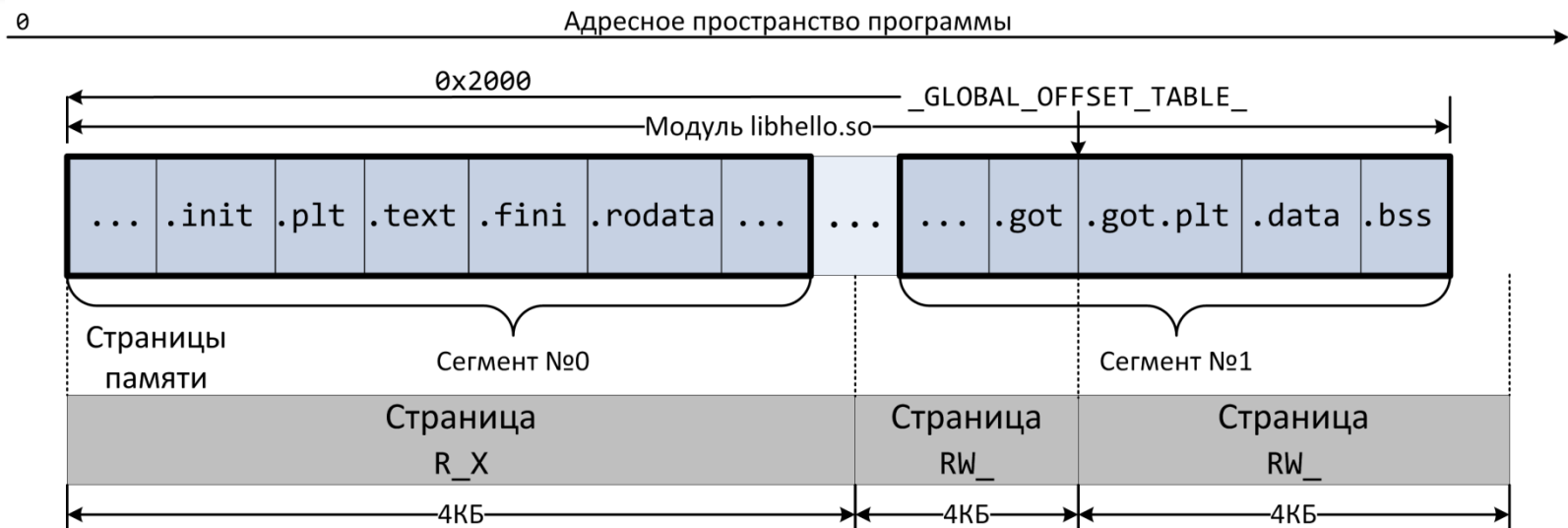
```
Section to Segment mapping:
```

```
Segment Sections...
```

```
00 ... .init .plt .text .fini .rodata ...
```

```
01 ... .got .got.plt .data .bss
```

```
...
```



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 базовый адрес GOT

```
snoop@jezek:~/samples/2017/linking$ objdump -d -r -M intel hello2_pic.o
hello2_pic.o:          file format elf32-i386
```

Disassembly of section .text:

```
00000000 <func>:
```

```
 0:  53 push    ebx
 1:  e8 fc ff ff ff call   1f <func+0x1f>
 2:  R_386_PC32    __x
 6:  81 c3 02 00 00
 8:  R_386_GOTPC   __GL
 c:  83 ec 10     sub
 f:  8b 83 00 00 00
11:  R_386_GOT32   buf
15:  ff 30     push
17:  8d 83 00 00 00
19:  R_386_GOTOFF  .L
1d:  50 push    eax
1e:  e8 fc ff ff ff call   1f <func+0x1f>
1f:  R_386_PLT32   printf
23:  83 c4 18     add    esp,0x18
26:  5b pop     ebx
27:  c3 ret
```

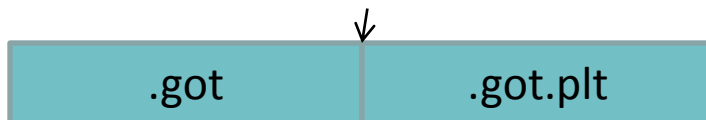
- Тип перебазирувания R_386_PLT **новое значение ссылки = $L + A - P$**
- L – абсолютный адрес элемента PLT, используемого для вызова заданного символа
- A – дополнительное слагаемое (addend), хранимое непосредственно в байтах ссылки
- P – абсолютный адрес ссылки
- Наличие такого типа перебазирувания в файле требует от компоновщика создавать PLT и дополнительный элемент в GOT
- Новое значение ссылки – смещение от данной инструкции до соответствующего элемента PLT

```

noop@jezek:~/samples/2017/linking$ objdump -d -M intel libhello.so
...
00000400 <printf@plt>:
 400:  ff a3 0c 00 00 00 jmp     DWORD PTR [ebx+0xc]
 406:  68 00 00 00 00  push   0x0
 40b:  e9 e0 ff ff ff   jmp     3f0 <_init+0x30>
...
00000570 <func>:
 570:  53  push   ebx
 571:  e8 ba fe ff ff   call   430 <__x86.get_pc_thunk.bx>
 576:  81 c3 8a 1a 00 00 add    ebx,0x1a8a
 57c:  83 ec 10  sub    esp,0x10
 57f:  8b 83 f4 ff ff ff mov    eax,DWORD PTR [ebx-0xc]
 585:  ff 30  push  DWORD PTR [eax]
 587:  8d 83 ac e5 ff ff lea   eax,[ebx-0x1a54]
 58d:  50  push  eax
 58e:  e8 6d fe ff ff   call   400 <printf@plt>
 593:  83 c4 18  add    esp,0x18
 596:  5b  pop    ebx
 597:  c3  ret

```

_GLOBAL_OFFSET_TABLE_



Первый вызов функции printf

```
noop@jezek:~/samples/2017/linking$ objdump -d -M intel libhello.so
```

```
...
```

```
Disassembly of section .plt:
```

```
000003f0 <printf@plt-0x10>:
```

```
3f0:  ff b3 04 00 00 00  push   DWORD PTR [ebx+0x4]
3f6:  ff a3 08 00 00 00  jmp     DWORD PTR [ebx+0x8]
3fc:  00 00             add    BYTE PTR [eax],al
```

```
...
```

```
00000400 <printf@plt>:
```

```
400:  ff a3 0c 00 00 00  jmp     DWORD PTR [ebx+0xc]
406:  68 00 00 00 00    push   0x0
40b:  e9 e0 ff ff ff    jmp     3f0 <_init+0x30>
```

`_GLOBAL_OFFSET_TABLE_`

`+4`

`+8`

`+c`

Динамический компоновщик

Описание модуля
libhello.so

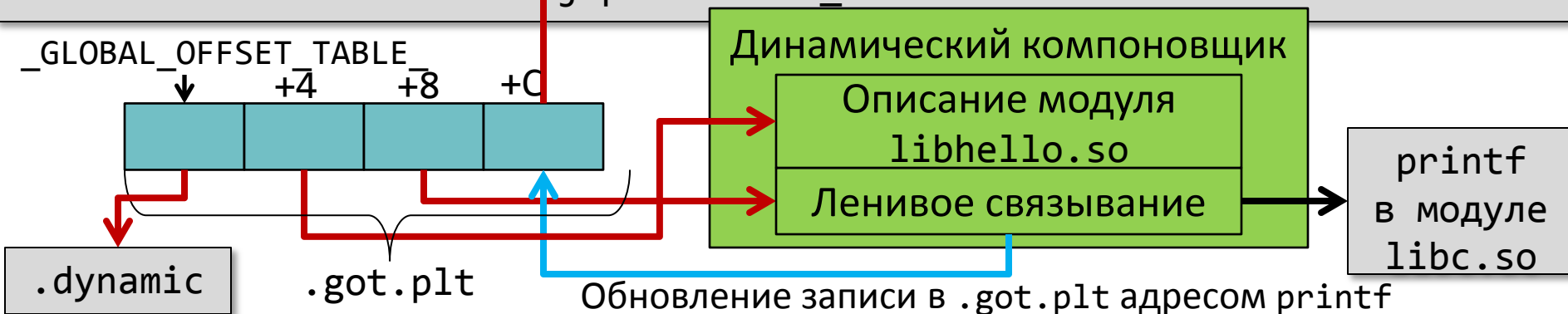
Ленивое связывание

printf
в модуле
libc.so

`.dynamic`

`.got.plt`

Обновление записи в `.got.plt` адресом `printf`



```
snoop@jezek:~/samples/2017/linking$ readelf -l hello-dlib
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x8048481
```

```
There are 9 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x00708	0x00708	R E	0x1000
LOAD	0x000f00	0x08049f00	0x08049f00	0x00124	0x00128	RW	0x1000
DYNAMIC	0x000f0c	0x08049f0c	0x08049f0c	0x000f0	0x000f0	RW	0x4

```
...
```

```
Section to Segment mapping:
```

```
Segment Sections...
```

```
...
```

02	.interpplt	.text	.fini	.rodata	...
03	.init_array	.fini_array	.jcr	.dynamic	.got	.got.plt .data .bss
04	.dynamic					

```
...
```

Загрузка динамически скомпонованного исполняемого файла

```
snoop@jezek:~/samples/2017/linking$ ldd hello-dlib
linux-gate.so.1 => (0xb772d000)
hello.so => /home/snoop/lib/hello.so (0xb7725000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb754f000)
/lib/ld-linux.so.2 (0x80077000)
```

- Начальная часть загрузки происходит аналогично, но на `_start` не «идем»
- В секции `.interp` хранится имя программы-интерпретатора, т.е. динамического компоновщика.
Указанный в секции интерпретатор загружается в память.
 - `ld-linux.so` скомпонован полностью статически
- Компоновщику передается адрес секции `.dynamic`, где собраны описания зависимостей, таблиц символов и других служебных данных, используемых в динамической компоновке.
 - Содержимое `GOT`, относящееся к переменным, обновляется при загрузке
- Необходимые динамические библиотеки ищутся по списку директорий из переменной окружения `LD_LIBRARY_PATH`
- Отработавший динамический компоновщик передает управление на точку входа

```
snoop@jezek:~/samples/2017/linking$ cp libhello.so $HOME/lib
snoop@jezek:~/samples/2017/linking$ export LD_LIBRARY_PATH=$HOME/lib
snoop@jezek:~/samples/2017/linking$ ./hello-dlib
Hello, world!
```

Далее: Аппаратура ЭВМ

- Организация аппаратного обеспечения компьютера
 - Физические основы, шины, периферийные устройства
- Организация памяти
 - НЖМД, кэш
 - Производительность
- Организация ЦПУ
 - Конвейер, система команд
- Многозадачная работа компьютера
 - Изоляция пользовательских программ