Лекция 0x13

19 апреля

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

```
student@pc:~/asm/linking$ gcc -m32 -static -o hello-static hello1.o hello2.o
student@pc:~/asm/linking$ gcc -m32 -o hello hello1.o hello2.o
student@pc:~/asm/linking$ ls -l hello-static hello
-rwxr-xr-x 1 student student   7248 апр 14 13:57 hello
-rwxr-xr-x 1 student student 657672 апр 14 13:57 hello-static
```

- Способ преодолеть эти недостатки: динамические библиотеки (shared libraries)
  - Объектные файлы, в которых содержатся код и данные компонуются с приложением *динамически*, либо во время *загрузки*, либо во время *выполнения*
  - Практикуются названия: DLL-ки, `.so`-шники

Drepper U. How To Write Shared Libraries. — 2011. https://www.akkadia.org/drepper/dsohowto.pdf

Динамические (разделяемые) библиотеки

- Динамическая компоновка происходит при запуске программы, когда исполняемый файл 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 (работающими программами).

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

Makefile

```
OBJS = hello-dlib hello-static libhello.so hello

CFLAGS = -m32 -O3 -D_FORTIFY_SOURCE=0 -fno-asynchronous-unwind-tables

all: $(OBJS)

hello-dlib: hello1.o libhello.so
	gcc -m32 -o hello-dlib hello1.o libhello.so

libhello.so: hello2_pic.o
	gcc -m32 -shared -o libhello.so hello2_pic.o

hello2_pic.s: hello.h hello2.c
	gcc -S -masm=intel $(CFLAGS) -fPIC -o hello2_pic.s hello2.c

hello2_pic.o: hello2_pic.s
	gcc -c $(CFLAGS) -o hello2_pic.o hello2_pic.s

hello1.o: hello.h hello1.c
	gcc -c $(CFLAGS) -fno-PIC -o hello1.o hello1.c

hello2.o: hello.h hello2.c
	gcc -c $(CFLAGS) -fno-PIC -o hello2.o hello2.c

hello-static: hello.h hello1.o hello2.o
	gcc $(CFLAGS) -static -fno-PIC -no-pie -o hello-static hello1.o hello2.o

hello: hello.h hello1.c hello2.c
	gcc $(CFLAGS) -o hello hello1.c hello2.c

clean:
	rm -f $(OBJS) hello2_pic.s
```

gcc version 7.5.0

hello.h

```
extern void f();

extern char* buf;
```

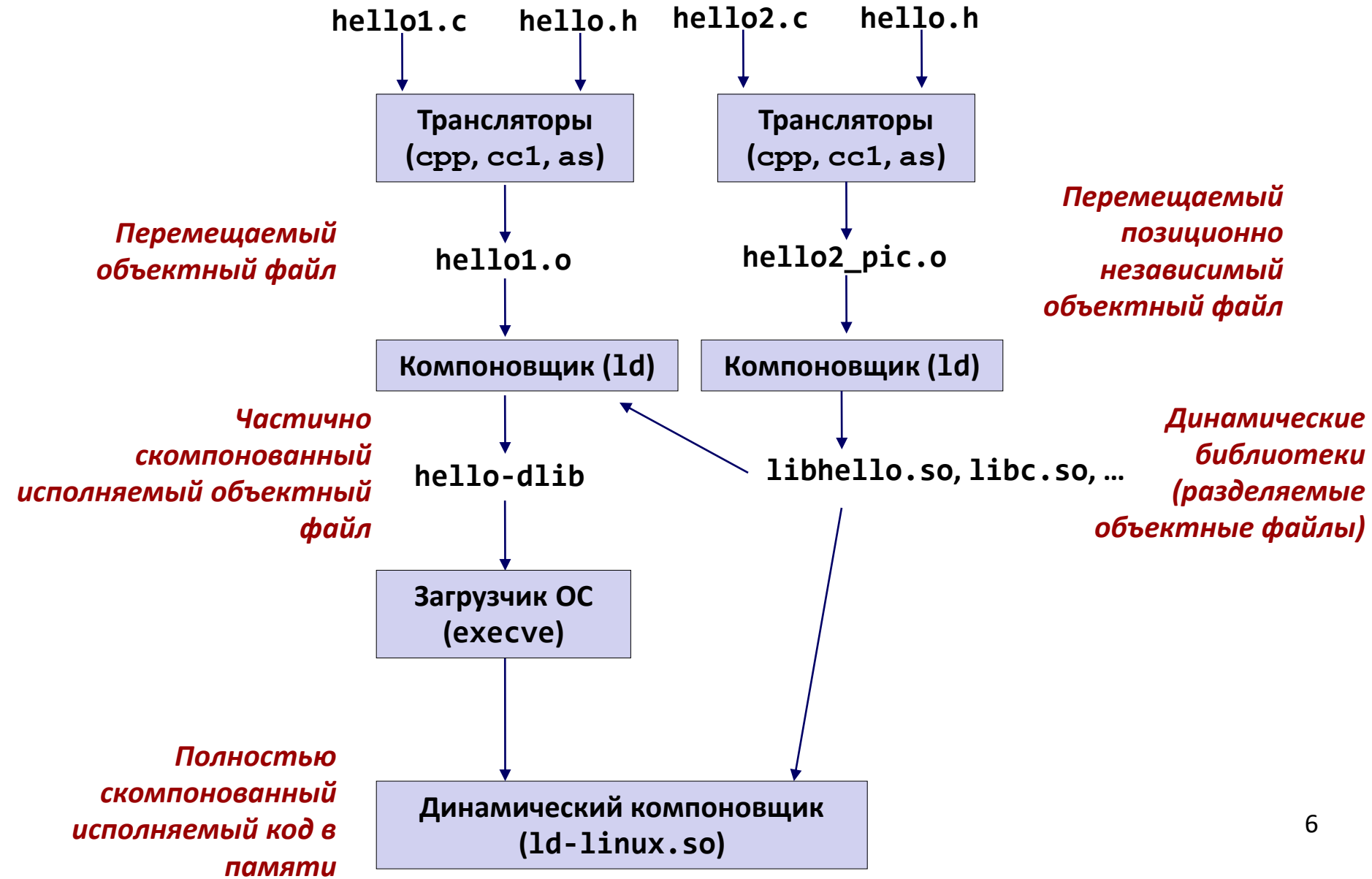
hello1.c

```
#include "hello.h"

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

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

hello2.c

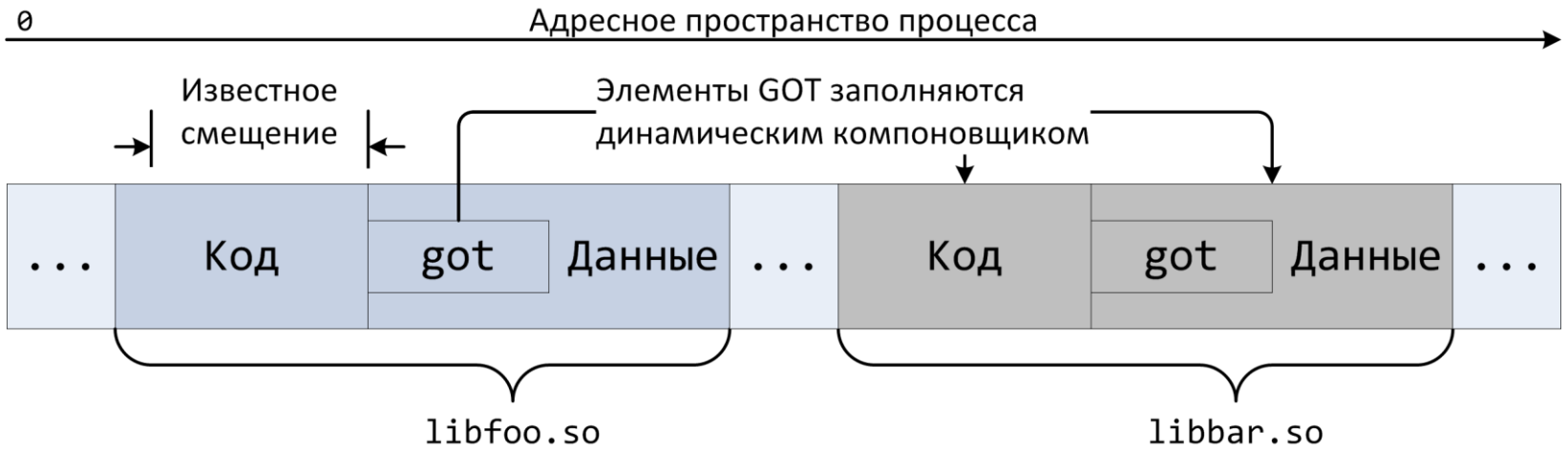
```
#include <stdio.h>
#include "hello.h"

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

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 относительного 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?

hello.h

```
extern void f();

extern char* buf;
```

hello2.c

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

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

Синтаксис ассемблера gas оставлен без изменений.

```
f:
   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***

```
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 <f>:
   0:   53                      push   ebx
   1:   e8 fc ff ff ff          call   2 <f+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_GOT32X 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 <f+0x1f>
                        1f: R_386_PLT32 printf
  23:   83 c4 18                add    esp,0x18
  26:   5b                      pop    ebx
  27:   c3                      ret
```

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

```
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 <f>:
   0:   53                      push   ebx
   1:   e8 fc ff ff ff          call   2 <f+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 0
  15:   ff 30
  17:   8d 83 00 00 0
  1d:   50
  1e:   e8 fc ff ff f
  23:   83 c4 18
  26:   5b
  27:   c3                      ret
```

- Тип перебазирования 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 перебазирования 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 fo



Disassembly of section .t

00000000 <f>:
   0:   53                      
   1:   e8 fc ff ff ff          
                        2
   6:   81 c3 02 00 00 00
                        8
   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 <f+0x1f>
                        1f: R_386_PLT32 printf
  23:   83 c4 18                        add    esp,0x18
  26:   5b                              pop    ebx
  27:   c3                              ret
```

- Тип перебазирования R_386_GOT32 ***новое значение ссылки = G + A***
- *G* – смещение в GOT, по которому должен быть размещен адрес символа
- *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 необходимого элемента

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


Disassembly of section .t

00000000 <f>:
   0:   53
   1:   e8 fc ff ff ff
                        2
   6:   81 c3 02 00 00 00
                        8
   c:   83 ec 10
   f:   8b 83 00 00 00 00
                        1
  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 <f+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
...
000004b0 <f>:
 4b0:   53                      push   ebx
 4b1:   e8 fa fe ff ff          call   3b0 <__x86.get
 4b6:   81 c3 4a 1b 00 00       add    ebx,0x1b4a
 4bc:   83 ec 10                sub    esp,0x10
 4bf:   8b 83 f8 ff ff ff       mov    eax,DWORD PTR [ebx-0x8]
 4c5:   ff 30                   push   DWORD PTR [eax]
 4c7:   8d 83 ec e4 ff ff       lea    eax,[ebx-0x1b14]
 4cd:   50                      push   eax
 4ce:   e8 bd fe ff ff          call   390 <printf@plt>
 4d3:   83 c4 18                add    esp,0x18
 4d6:   5b                      pop    ebx
 4d7:   c3                      ret
```

Расстояние до GOT

Смещение от GOT до элемента, соответствующего указателю на buf

Смещение от GOT до места в памяти, где размещена строка .LC0

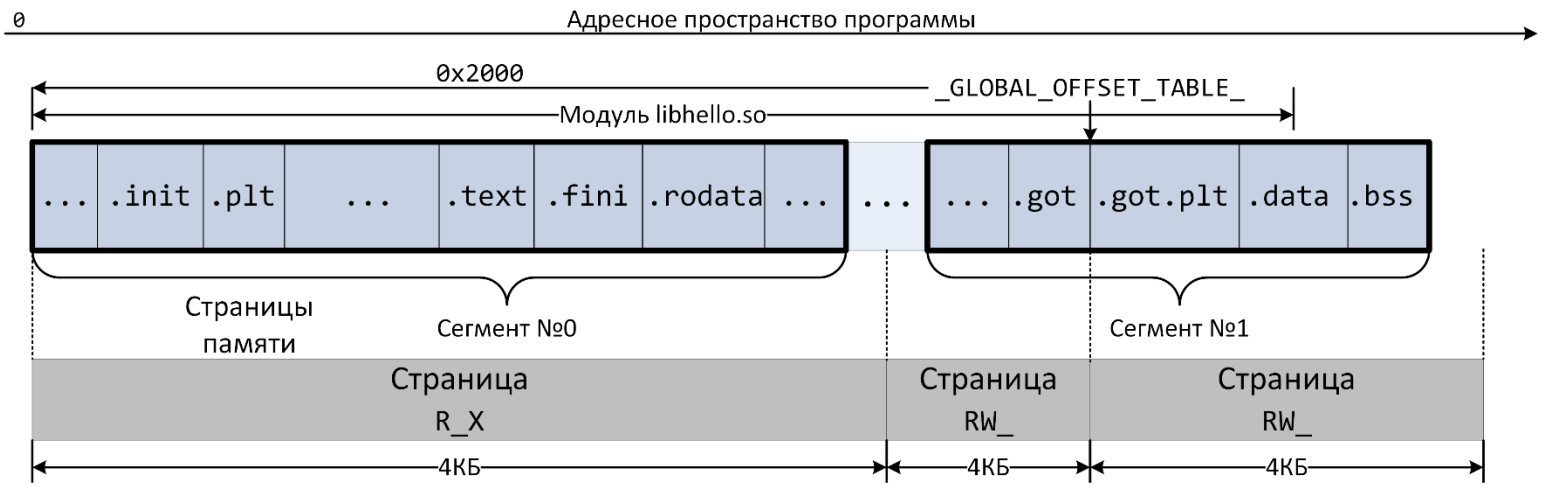
- Компоновщик создает GOT
- Все ссылки в коде обновляются
- Создается секция .dynsym, в которой собраны описания символов, используемых в динамической компоновке
- Создается секция .rel.dyn, в которой собраны описания новых ссылок, размещенных в GOT

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

Symbol table '.dynsym' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
     5: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
...
     8: 000004b0    40 FUNC    GLOBAL DEFAULT   12 f

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
...
00001ff8  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                    ........
```

- Тип перебазирования R_386_GLOB_DAT ***новое значение ссылки = S***
- *S* – абсолютный адрес памяти, связанный с символом

_GLOBAL_OFFSET_TABLE_ - 0x8 = 0x2000 – 0x8 = 0x1ff8

```
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 0x004f4 0x004f4 R E 0x1000
LOAD      0x000f04 0x00001f04 0x00001f04 0x00110 0x00114 RW  0x1000
...
 Section to Segment mapping:
  Segment Sections...
   00     ... .init .plt ... .text .fini .rodata ...
   01     ... .got .got.plt .data .bss
...
```



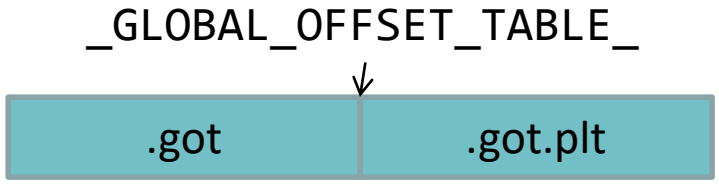
Procedure Linkage Table

- 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 <f>:
   0:   53
   1:   e8 fc ff ff ff
   6:   81 c3 02 00 00
   c:   83 ec 10
   f:   8b 83 00 00 00
  15:   ff 30
  17:   8d 83 00 00 00
  1d:   50                      push   eax
  1e:   e8 fc ff ff ff          call   1f <f+0x1f>
                        1f: R_386_PLT32       printf
  23:   83 c4 18                add    esp,0x18
  26:   5b                      pop    ebx
  27:   c3                      ret
```

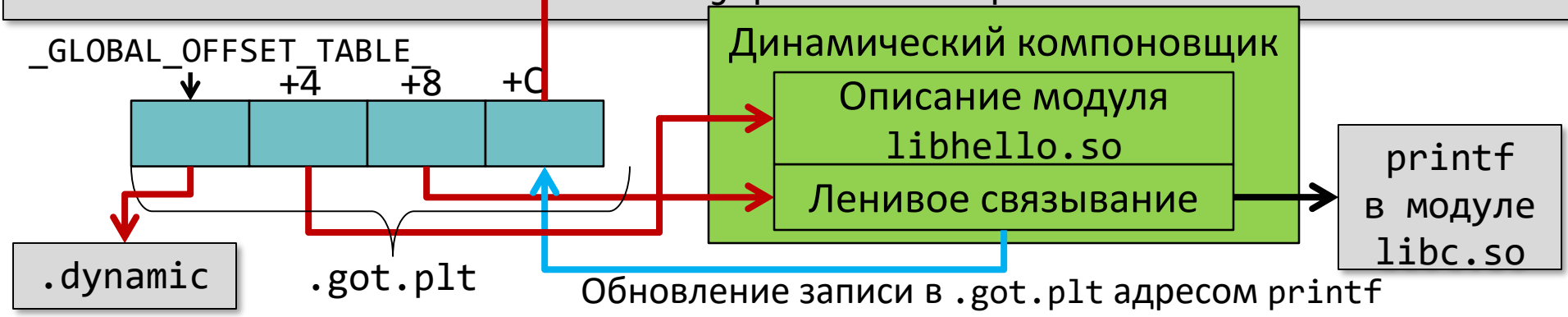
- Тип перебазирования R_386_PLT
  новое значение ссылки = L + A – P
- *L* – абсолютный адрес элемента PLT, используемого для вызова заданного символа
- *A* – дополнительное слагаемое (addend), хранимое непосредственно в байтах ссылки
- P – абсолютный адрес ссылки
- Наличие такого типа перебазирования 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
...
00000390 <printf@plt>:
 390:   ff a3 0c 00 00 00       jmp    DWORD PTR [ebx+0xc]
 396:   68 00 00 00 00          push   0x0
 39b:   e9 e0 ff ff ff          jmp    380 <.plt>
...
000004b0 <f>:
 4b0:   53                      push   ebx
 4b1:   e8 fa fe ff ff          call   3b0 <__x86.get_pc_thunk.bx>
 4b6:   81 c3 4a 1b 00 00       add    ebx,0x1b4a
 4bc:   83 ec 10                sub    esp,0x10
 4bf:   8b 83 f8 ff ff ff       mov    eax,DWORD PTR [ebx-0x8]
 4c5:   ff 30                   push   DWORD PTR [eax]
 4c7:   8d 83 ec e4 ff ff       lea    eax,[ebx-0x1b14]
 4cd:   50                      push   eax
 4ce:   e8 bd fe ff ff          call   390 <printf@plt>
 4d3:   83 c4 18                add    esp,0x18
 4d6:   5b                      pop    ebx
 4d7:   c3                      ret
```



Первый вызов функции printf

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

00000380 <.plt>:
 380:   ff b3 04 00 00 00     push   DWORD PTR [ebx+0x4]
 386:   ff a3 08 00 00 00     jmp    DWORD PTR [ebx+0x8]
 38c:   00 00                 add    BYTE PTR [eax],al
        ...

00000390 <printf@plt>:
 390:   ff a3 0c 00 00 00     jmp    DWORD PTR [ebx+0xc]
 396:   68 00 00 00 00        push   0x0
 39b:   e9 e0 ff ff ff        jmp    380 <.plt>
```



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

Elf file type is DYN (Shared object file)
Entry point 0x3b0
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x004f4 0x004f4 R E 0x1000
  LOAD           0x000f04 0x00001f04 0x00001f04 0x00110 0x00114 RW  0x1000
  DYNAMIC        0x000f0c 0x00001f0c 0x00001f0c 0x000e0 0x000e0 RW  0x4
  NOTE           0x0000f4 0x000000f4 0x000000f4 0x00024 0x00024 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f04 0x00001f04 0x00001f04 0x000fc 0x000fc R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gu.build-id .gnu.hash .dynsym .dynstr .gnu.version
.gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini .rodata
.eh_frame
   01     .init_array .fini_array .dynamic .got .got.plt .data .bss
   02     .dynamic
   03     .note.gu.build-id
   04
   05     .init_array .fini_array .dynamic .got
```

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

Elf file type is DYN (Shared object file)
Entry point 0x4c8
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x007ac 0x007ac R E 0x1000
  LOAD           0x000ed0 0x00001ed0 0x00001ed0 0x0013c 0x00140 RW  0x1000
  DYNAMIC        0x000ed8 0x00001ed8 0x00001ed8 0x00100 0x00100 RW  0x4
...
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
...
```

Загрузка динамически скомпонованного исполняемого файла

```
student@pc:~/asm/linking$ LD_LIBRARY_PATH=`pwd` ldd hello-dlib
        linux-gate.so.1 (0xf7fd5000)
        libhello.so => /home/student/asm/linking/libhello.so (0xf7fca000)
        libc.so.6 => /lib32/libc.so.6 (0xf7ddd000)
        /lib/ld-linux.so.2 (0xf7fd6000)
```

- 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

```
student@pc:~/asm/linking$ export LD_LIBRARY_PATH=/home/student/asm/linking
student@pc:~/asm/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