

Санкт-Петербургский политехнический университет Петра Великого
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчёт по практической работе № 1
по предмету «Проектирование ОС и компонентов»

ЗАГРУЗКА ПРИЛОЖЕНИЙ (WINDOWS/LINUX)

Работу выполнил студент гр. 63501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Санкт-Петербург
2016

Содержание

Постановка задачи	3
Введение	4
1 Процесс загрузки приложений в Linux	5
1.1 ELF – формат исполнения и компоновки	5
1.2 Сегменты и секции	6
1.3 Структура и назначение полей служебных заголовков	7
1.4 Процесс загрузки в память	8
1.5 Резидентное приложение – монитор сетевой активности	9
1.6 Динамические библиотеки .so	18
1.7 Резидентное приложение с динамической библиотекой	19
2 Процесс загрузки приложений Windows	21
2.1 Выполнение EXE-модуля	21
2.2 Динамически подключаемые библиотеки	22
2.3 Реализация резидентного приложения	24
2.4 Анализ исполнения приложения	27
Заключение	31
Список литературы	32

Постановка задачи

В рамках данной работы необходимо написать полезную программу для ОС семейства Linux и Windows. Программа должна быть выполнена в качестве резидентного (не демона) приложения. Далее переписать ту же программу с использованием динамически загружаемой библиотеки.

Таким образом, в результате работы должно получиться четыре программы:

- Резидентное приложение для Windows собранное единым модулем
- Резидентное приложение для Windows с динамической библиотекой (.dll)
- Резидентное приложение для Linux собранное единым модулем
- Резидентное приложение для Linux с динамической библиотекой (.so)

В процессе работы требуется изучить принцип загрузки приложений в различных операционных системах и описать особенности приложений использующих динамические библиотеку.

Введение

В связи с тем, что сегодня уровень сложности программного обеспечения очень высок, разработка приложений с использованием только какого-либо языка программирования (например, языка C) значительно затрудняется. Программист должен затратить массу времени на решение стандартных задач по созданию многооконного интерфейса. Реализация технологии связывания и встраивания объектов потребует от программиста еще более сложной работы.

Чтобы облегчить работу программиста, следует пере использовать ранее написанный код. С одной стороны это позволяет не решать одну задачу дважды, с другой – появляются дополнительные требования по оформлению существующих решений и их распространению. Долгое время в Unix (а потом Linux) среде распространение велось с исходных кодах. При этом предполагалось, что пользователь достаточно грамотен для работы с таким источником.

С распространением персональных компьютеров и приложений для них, получили популярность динамические библиотеки, которые позволяли, в частности, обновлять приложения без их пере сборки конечным пользователем, использовать различные языки для решения разных задач и даже упрощение локализации.

В данной работе рассматривается процесс загрузки приложений на операционных системах семейства Windows и Linux и порядок работы с динамическими библиотеками.

1 Процесс загрузки приложений в Linux

1.1 ELF – формат исполнения и компоновки

Изначально UNIX (и производные от нее операционные системы) поддерживали множество исполняемых форматов, но теперь стандартом де-факто для LINUX и BSD стал ELF. Стандарт для формата ELF изначально был разработан и опубликован компанией USL как часть двоичного интерфейса приложений операционной системы UNIX System V. Затем он был выбран комитетом TIS и развит в качестве переносимого формата для различных операционных систем, работающих на 32-разрядной аппаратной архитектуре Intel x86. ELF быстро набрал популярность и, после того как компания HP расширила формат и опубликовала стандарт ELF-64, распространился и на 64-разрядных платформах. Иногда еще встречается древний a.out, но это достаточно особые случаи, требующие совместимости с железом.

Аббревиатура ELF расшифровывается как Execution and Linkable Format (формат исполнения и компоновки). Он во многом напоминает win32 PE. В начале ELF-файла расположен служебный заголовок (ELF-header), описывающий основные характеристики файла — тип (исполнения или линковки), архитектура ЦП, виртуальный адрес точки входа, размеры и смещения остальных заголовков. . .

За ELF-header'ом следует таблица сегментов (program header table), перечисляющая имеющиеся сегменты и их атрибуты. В формате линковки она необязательна. Линкеру сегменты не важны и он работает исключительно на уровне секций. Напротив, системный загрузчик, загружающий исполняемый ELF-файл в память, игнорирует секции, и оперирует целыми сегментами[1].

Стандарт формата ELF различает несколько типов файлов:

- Перемещаемый файл – хранит инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такой связи может быть разделяемый объектный файл или исполняемый файл. К этому типу относятся объектные файлы статических библиотек.
- Разделяемый объектный файл – также содержит инструкции и данные и может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан новый объектный файл, либо при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.

- Исполняемый файл – содержит полное описание, позволяющее системе создать образ процесса. В том числе: инструкции, данные, описание необходимых разделяемых объектных файлов и необходимую символьную и отладочную информацию.

1.2 Сегменты и секции

Сегмент – это непрерывная область адресного пространства со своими атрибутами доступа. В частности, сегмент кода имеет атрибут исполнения, а сегмент данных – атрибуты чтения и записи. Стоит отметить, что ELF-сегменты это не сегменты x86 процессора! В защищенном режиме 386+ никаких "сегментов" уже нет, а есть только селекторы и все сегменты ELF-файла загружаются в единый 4 Гбайтовый x86-сегмент! В зависимости от типа сегмента, величина выравнивания в памяти может варьировать от 4h до 1000h байт (размер страницы на x86). В самом ELF-файле хранятся в невыровненном виде, плотно прижатые друг к другу.

Ближайший аналог ELF-сегментов – PE-секции, но в PE-файлах, секция – это наименьшая структурная единица, а в ELF-файлах сегмент может быть разбит на один или несколько фрагментов – секций. В частности, типичный кодовый сегмент состоит из

- секций `.init` – процедуры инициализации,
- секции `.plt` – секция связок,
- секции `.text` – основной код программы,
- секции `.fini` – процедуры финализации.

Секции нужны линкеру для комбинирования, чтобы он мог отобрать секции с похожими атрибутами и оптимальным образом растасовать их по сегментам при сборке файла, то есть "скомбинировать" [2].

Несмотря на то, что системный загрузчик игнорирует таблицу секций, линкер все-таки помещает ее копию в исполняемый файл. Это приводит к не значительному расходу места, зато эта информация полезна для отладчиков и дизассемблеров. По не совсем понятным причинам `gdb` и многие другие программы отказываются загружать в файл с поврежденной или отсутствующей таблицей секций, чем часто пользуются для защиты программ от постороннего вмешательства. Структура файла представлена на рисунке 1.

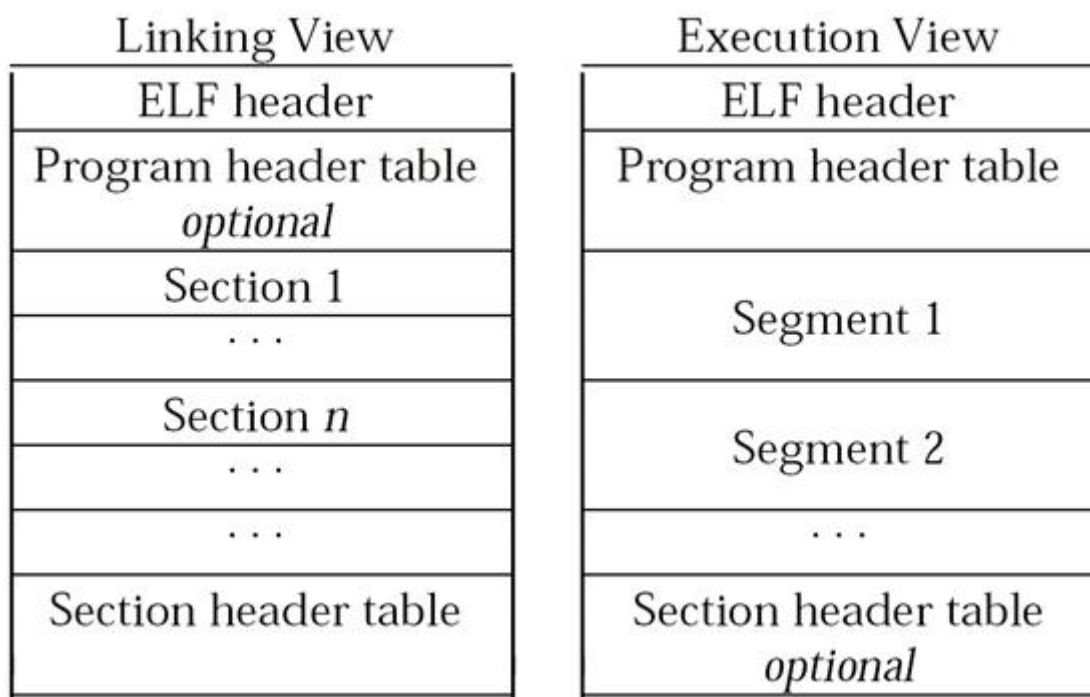


Рис. 1: Структура ELF-формат с точки зрения линкера (слева) и системного загрузчика операционной системы (справа)

1.3 Структура и назначение полей служебных заголовков

Заголовок файла (ELF Header) имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры и смещения остальных частей файла.

- *e_ident[]* – Массив байт, каждый из которых определяет общую характеристику файла. Первые четыре байта в массиве определяют сигнатуру файла и всегда должны содержать 0x7f 0x45 0x4c 0x46 соответственно.
- *e_type* – Тип файла.
- *e_machine* – Архитектура аппаратной платформы, для которой файл создан.
- *e_version* – Номер версии формата.
- *e_entry* – Точка входа.
- *e_phoff* – Расположение таблицы заголовков программы.
- *e_shoff* – Расположение таблицы заголовков разделов.
- *e_flags* – Связанные с файлом флаги, зависящие от процессора.

- *e_ehsize* – Размер[5] заголовка файла.
- *e_phentsize* – Размер каждого заголовка программы.
- *e_phnum* – Число заголовков программы.
- *e_shentsize* – Размер каждого заголовка разделов.
- *e_shnum* – Число заголовков разделов.
- *e_shstrndx* – Индекс записи в таблице разделов, указывающей на таблицу названий разделов.

1.4 Процесс загрузки в память

По умолчанию ELF-заголовок проецируется по адресу 8048000h, который прописан в его заголовке. Это и есть базовый адрес загрузки. На стадии линковки он может быть свободно изменен на другой, но большинство программистов оставляют его "как есть". Все сегменты проецируются в память в соответствии с виртуальными адресами, прописанными в таблице сегментов, причем, виртуальная проекция образа всегда непрерывна, и между сегментами не должно быть незаполненных "дыр".

Начиная с адреса 40000000h располагаются совместно используемые библиотеки ld-linux.so, libm.so, libc.so и другие, которые связывают операционную систему с прикладной программой. Ближайший аналог из мира Windows – KERNL32.DLL, реализующая win32 API, что расшифровывается как Application Programming Interface, но при желании программа может вызывать функции операционной системы и напрямую. В NT за это отвечает прерывание INT 2Eh, в LINUX – как правило INT 80h (на самом деле к текущему моменту в этом вопросе была проделана некоторая оптимизация, о которой будет сказано позже, при рассмотрении вывода утилиты ldd)[3].

Для вызова функций типа открытия файла мы можем обратиться либо к библиотеке libc, либо непосредственно к самой операционной системе. Первый вариант – самый громоздкий, самый переносимый, и наименее приметный. Последний – прост в реализации, но испытывает проблемы совместимости с различными версиями LINUX'a.

Последний гигабайт адресного пространства (от адреса C0000000h и выше) занимают код и данные операционной системе, к которым можно обращаться только посредством прерывания INT 80h или через разделяемые библиотеки.

Стек находится в нижних адресах. Он начинается с базового адреса загрузки и "растет вверх" по направлению к нулевым адресам. В большинстве Линукс-систем стек исполняем

(то есть сюда можно скопировать машинный код и передать на него управления), однако, некоторые администраторы устанавливают заплатки, отнимающие у стека атрибут исполнимости. Карта памяти представлена на рисунке 2.

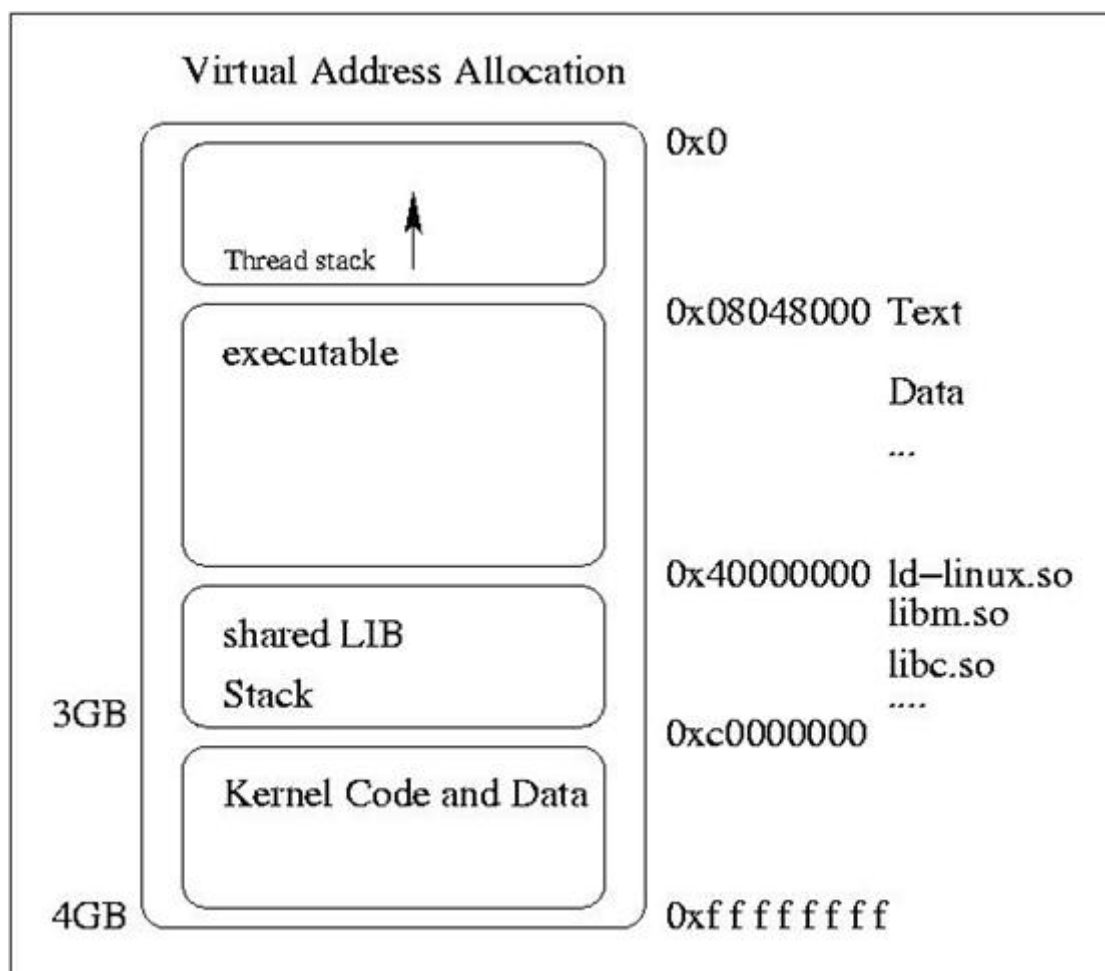


Рис. 2: Карта памяти загруженного образа исполняемого файла

1.5 Резидентное приложение – монитор сетевой активности

В качестве полезного приложения было решено создать простую утилиту, которая отображает количество полученных и отправленных пакетов по указанному сетевому интерфейсу. Процесс организации интерфейса с пользователем интереса не представляет, но работа с системой построена по средствам извлечения информации из файла `/proc/net/dev` и представлена в листинге 1.

Листинг 1: Функция получения информации о трафике по сетевому интерфейсу (src/ELF/lin/parse.cpp)

```

7 bool parse(char* ifname, long long* rx_bytes, long long* rx_packets,
8           long long* tx_bytes, long long* tx_packets) {

```

```

9      std::string interface(ifname);
10     interface.append(":");
11     std::string buff;
12     std::ifstream netstat("/proc/net/dev");
13
14     while(std::getline(netstat, buff)) {
15         size_t shift = buff.find_first_not_of(' ');
16         if (buff.compare(shift, interface.length(), interface) == 0)
17             {
18                 std::regex rx(R"([^\[:alpha:]]+[:digit:]+[^\[:alpha:]])");
19                 ;
20                 std::sregex_iterator pos(buff.cbegin(), buff.cend(), rx)
21                 ;
22
23                 *rx_bytes = std::stoll(pos->str());
24                 ++pos;
25                 *rx_packets = std::stoll(pos->str());
26                 std::advance(pos, 7);
27                 *tx_bytes = std::stoll(pos->str());
28                 ++pos;
29                 *tx_packets = std::stoll(pos->str());
30
31                 return true;
32             }
33     }
34     return false;
35 }

```

После компиляции можно собрать информацию об объектном файле. Полная демонстрация возможностей objdump займёт довольно много места, но основные возможности представлены в следующем листинге 2. В 1-й строке запрашивается информация о хедерах файла, их именах и расположении.

Листинг 2: Демонстрация работы программы objdump

```

1 user@host$ objdump —headers ./netmonitor
2
3 ./netmonitor:      file format elf64-x86-64
4

```

5	Sections :					
6	Idx	Name	Size	VMA	LMA	File
		off	Algn			
7	0	.interp	0000001c	0000000000400238	0000000000400238	
		00000238	2**0			
8			CONTENTS, ALLOC, LOAD, READONLY, DATA			
9	1	.note.ABI-tag	00000020	0000000000400254	0000000000400254	
		00000254	2**2			
10			CONTENTS, ALLOC, LOAD, READONLY, DATA			
11	2	.note.gnu.build-id	00000024	0000000000400274	0000000000400274	
		00000274	2**2			
12			CONTENTS, ALLOC, LOAD, READONLY, DATA			
13	3	.gnu.hash	000000b4	0000000000400298	0000000000400298	
		00000298	2**3			
14			CONTENTS, ALLOC, LOAD, READONLY, DATA			
15	4	.dynsym	00000918	0000000000400350	0000000000400350	
		00000350	2**3			
16			CONTENTS, ALLOC, LOAD, READONLY, DATA			
17	5	.dynstr	00000e70	0000000000400c68	0000000000400c68	
		00000c68	2**0			
18			CONTENTS, ALLOC, LOAD, READONLY, DATA			
19	6	.gnu.version	000000c2	0000000000401ad8	0000000000401ad8	
		00001ad8	2**1			
20			CONTENTS, ALLOC, LOAD, READONLY, DATA			
21	7	.gnu.version_r	000000f0	0000000000401ba0	0000000000401ba0	
		00001ba0	2**3			
22			CONTENTS, ALLOC, LOAD, READONLY, DATA			
23	8	.rela.dyn	00000180	0000000000401c90	0000000000401c90	
		00001c90	2**3			
24			CONTENTS, ALLOC, LOAD, READONLY, DATA			
25	9	.rela.plt	000006f0	0000000000401e10	0000000000401e10	
		00001e10	2**3			
26			CONTENTS, ALLOC, LOAD, READONLY, DATA			
27	10	.init	0000001a	0000000000402500	0000000000402500	
		00002500	2**2			
28			CONTENTS, ALLOC, LOAD, READONLY, CODE			
29	11	.plt	000004b0	0000000000402520	0000000000402520	

```

00002520  2**4
30          CONTENTS, ALLOC, LOAD, READONLY, CODE
31 12 .plt.got      00000008  000000000004029d0  000000000004029d0
    000029d0  2**3
32          CONTENTS, ALLOC, LOAD, READONLY, CODE
33 13 .text        00011b42  000000000004029e0  000000000004029e0
    000029e0  2**4
34          CONTENTS, ALLOC, LOAD, READONLY, CODE
35 14 .fini        00000009  00000000000414524  00000000000414524
    00014524  2**2
36          CONTENTS, ALLOC, LOAD, READONLY, CODE
37 15 .rodata      00000d33  00000000000414540  00000000000414540
    00014540  2**5
38          CONTENTS, ALLOC, LOAD, READONLY, DATA
39 16 .eh_frame_hdr 0000058c  00000000000415274  00000000000415274
    00015274  2**2
40          CONTENTS, ALLOC, LOAD, READONLY, DATA
41 17 .eh_frame    00002734  00000000000415800  00000000000415800
    00015800  2**3
42          CONTENTS, ALLOC, LOAD, READONLY, DATA
43 18 .gcc_except_table 00000875  00000000000417f34  00000000000417f34
    00017f34  2**2
44          CONTENTS, ALLOC, LOAD, READONLY, DATA
45 19 .init_array  00000010  00000000000618de8  00000000000618de8
    00018de8  2**3
46          CONTENTS, ALLOC, LOAD, DATA
47 20 .fini_array  00000008  00000000000618df8  00000000000618df8
    00018df8  2**3
48          CONTENTS, ALLOC, LOAD, DATA
49 21 .jcr         00000008  00000000000618e00  00000000000618e00
    00018e00  2**3
50          CONTENTS, ALLOC, LOAD, DATA
51 22 .dynamic     000001f0  00000000000618e08  00000000000618e08
    00018e08  2**3
52          CONTENTS, ALLOC, LOAD, DATA
53 23 .got         00000008  00000000000618ff8  00000000000618ff8
    00018ff8  2**3

```

```

54          CONTENTS, ALLOC, LOAD, DATA
55 24 .got.plt      00000268  00000000000619000  00000000000619000
    00019000  2**3
56          CONTENTS, ALLOC, LOAD, DATA
57 25 .data        00000420  00000000000619280  00000000000619280
    00019280  2**5
58          CONTENTS, ALLOC, LOAD, DATA
59 26 .bss         00000648  000000000006196a0  000000000006196a0
    000196a0  2**5
60          ALLOC
61 27 .comment     0000002f  00000000000000000  00000000000000000
    000196a0  2**0
62          CONTENTS, READONLY
63 28 .debug_aranges 00000ab0  00000000000000000  00000000000000000
    000196cf  2**0
64          CONTENTS, READONLY, DEBUGGING
65 29 .debug_info  00082dbe  00000000000000000  00000000000000000  0001
    a17f  2**0
66          CONTENTS, READONLY, DEBUGGING
67 30 .debug_abbrev 000019f8  00000000000000000  00000000000000000  0009
    cf3d  2**0
68          CONTENTS, READONLY, DEBUGGING
69 31 .debug_line  00008cb8  00000000000000000  00000000000000000  0009
    e935  2**0
70          CONTENTS, READONLY, DEBUGGING
71 32 .debug_str   0007f0c7  00000000000000000  00000000000000000  000
    a75ed  2**0
72          CONTENTS, READONLY, DEBUGGING
73 33 .debug_loc   0004c7c9  00000000000000000  00000000000000000
    001266b4  2**0
74          CONTENTS, READONLY, DEBUGGING
75 34 .debug_ranges 00012670  00000000000000000  00000000000000000
    00172e7d  2**0
76          CONTENTS, READONLY, DEBUGGING

```

Другой удобной программой для вывода информации о ELF файле является readelf (вывод программы приведён в сокращенном виде, листинг 3).

Листинг 3: Демонстрация работы программы readelf

```

89 ELF Header:
90   Magic:    7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00
91   Class:                                ELF64
92   Data:                                2's complement, little endian
93   Version:                               1 (current)
94   OS/ABI:                                UNIX – GNU
95   ABI Version:                           0
96   Type:                                    EXEC (Executable file)
97   Machine:                               Advanced Micro Devices X86–64
98   Version:                               0x1
99   Entry point address:                   0x402c20
100  Start of program headers:               64 (bytes into file)
101  Start of section headers:              1635848 (bytes into file)
102  Flags:                                  0x0
103  Size of this header:                    64 (bytes)
104  Size of program headers:                56 (bytes)
105  Number of program headers:               9
106  Size of section headers:                64 (bytes)
107  Number of section headers:              39
108  Section header string table index: 36
109
110
111 Program Headers:
112   Type           Offset           VirtAddr           PhysAddr
113                   FileSiz          MemSiz             Flags   Align
114  PHDR              0x0000000000000040  0x0000000000400040  0
115                   x0000000000400040
116                   0x00000000000001f8  0x00000000000001f8  R E     8
117  INTERP            0x0000000000000238  0x0000000000400238  0
118                   x0000000000400238
119                   0x000000000000001c  0x000000000000001c  R       1
120   [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD                0x0000000000000000  0x0000000000400000  0
                   x0000000000400000
                   0x00000000000187a9  0x00000000000187a9  R E
                   200000

```

121	LOAD	0x00000000000018de8	0x000000000000618de8	0	
		x000000000000618de8			
122		0x000000000000008b8	0x00000000000000f00	RW	
		200000			
123	DYNAMIC	0x00000000000018e08	0x000000000000618e08	0	
		x000000000000618e08			
124		0x000000000000001f0	0x000000000000001f0	RW	8
125	NOTE	0x00000000000000254	0x000000000000400254	0	
		x000000000000400254			
126		0x00000000000000044	0x00000000000000044	R	4
127	GNU_EH_FRAME	0x00000000000015274	0x000000000000415274	0	
		x000000000000415274			
128		0x0000000000000058c	0x0000000000000058c	R	4
129	GNU_STACK	0x00000000000000000	0x00000000000000000	0	
		x00000000000000000			
130		0x00000000000000000	0x00000000000000000	RW	10
131	GNU_RELRO	0x00000000000018de8	0x000000000000618de8	0	
		x000000000000618de8			
132		0x00000000000000218	0x00000000000000218	R	1
133					
134	Section to Segment mapping:				
135	Segment Sections...				
136	00				
137	01	.interp			
138	02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame . gcc_except_table			
139	03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss			
140	04	.dynamic			
141	05	.note.ABI-tag .note.gnu.build-id			
142	06	.eh_frame_hdr			
143	07				
144	08	.init_array .fini_array .jcr .dynamic .got			

Информацию о символах можем получить при помощи утилиты ss (нас интересует функция

parse, строка 56, листинг 4).

Листинг 4: Демонстрация работы программы nm

```
40 0000000000414520 T __libc_csu_fini
41 00000000004144b0 T __libc_csu_init
42      U __libc_start_main@@GLIBC_2.2.5
43 00000000004029e0 T main
44      U memcmp@@GLIBC_2.2.5
45      U memcpy@@GLIBC_2.14
46      U memmove@@GLIBC_2.2.5
47      w __pthread_key_create
48 0000000000402c90 t register_tm_clones
49      U __stack_chk_fail@@GLIBC_2.4
50 0000000000402c20 T _start
51      U strchr@@GLIBC_2.2.5
52      U strlen@@GLIBC_2.2.5
53      U strtoll@@GLIBC_2.2.5
54 00000000006196a0 D __TMC_END__
55      U _Unwind_Resume@@GCC_3.0
56 0000000000403140 T _Z5parsePcPxS0_S0_S0_
57      U _ZdlPv@@GLIBCXX_3.4
58      U _ZdlPvm@@CXXABI_1.3.9
59 0000000000619cd8 u
    _ZGVZNKSt8__detail11_AnyMatcherINSt7__cxx1112regex_traitsIcEELb0ELb0ELb0
60 0000000000619cc8 u
    _ZGVZNKSt8__detail11_AnyMatcherINSt7__cxx1112regex_traitsIcEELb0ELb0ELb1
61 0000000000619cb8 u
    _ZGVZNKSt8__detail11_AnyMatcherINSt7__cxx1112regex_traitsIcEELb0ELb1ELb0
62 0000000000619ca8 u
    _ZGVZNKSt8__detail11_AnyMatcherINSt7__cxx1112regex_traitsIcEELb0ELb1ELb1
63 00000000004048e0 W
    _ZN9__gnu_cxx6__stoaIxxcIiEEET0_PFT_PKT1_PPS3_DpT2_EPKcS5_PmS9_
64 00000000004048e0 W
    _ZN9__gnu_cxx6__stoaIxxcJiEEET0_PFT_PKT1_PPS3_DpT2_EPKcS5_PmS9_
```



```

65      U _ZNKSt5ctypeIcE13_M_widen_initEv@@GLIBCXX_3.4.11
66 0000000000403cd0 W _ZNKSt5ctypeIcE8do_widenEc
67 0000000000403ce0 W _ZNKSt5ctypeIcE9do_narrowEc
68      U _ZNKSt6locale2id5_M_idEv@@GLIBCXX_3.4

```

Зависимость от библиотек показывает утилита ldd (листинг 5). Тут стоит обратить внимание на виртуальную библиотеку в строке 1. В те времена, когда процессоры с архитектурой x86 только появились, взаимодействие пользовательских приложений со службами операционной системы осуществлялось с помощью прерываний. По мере создания более мощных процессоров эта схема взаимодействия становилась узким местом системы. Во всех процессорах, начиная с Pentium II, Intel реализовала механизм быстрых системных вызовов (Fast System Call), в котором вместо прерываний используются инструкции SYSENTER и SYSEXIT, ускоряющие выполнение системных вызовов.

Библиотека linux-vdso.so.1 является виртуальной библиотекой, или виртуальным динамически разделяемым объектом (VDSO), который размещается только в адресном пространстве отдельной программы. В более ранних системах эта библиотека называлась linux-gate.so.1. Эта виртуальная библиотека содержит всю необходимую логику, обеспечивающую для пользовательских приложений наиболее быстрый доступ к системным функциям в зависимости от архитектуры процессора – либо через прерывания, либо (для большинства современных процессоров) через механизм быстрых системных вызовов.

Листинг 5: Демонстрация работы программы ldd

```

1  linux-vdso.so.1 => (0x00007ffdl1a560000)
2  libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0
   x00007fbfaf1a0000)
3  libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0
   x00007fbfaef8a000)
4  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbfaebc0000)
5  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fbfae8b7000)
6  /lib64/ld-linux-x86-64.so.2 (0x0000562b8f3ca000)

```

Адрес загрузки программы не меняется, однако адреса подключения динамических библиотек и область размещения стека изменяются при повторном запуске. Отсюда можно сделать вывод о том, что представленные адреса виртуальные.

Теперь программу можно запустить для проверки сетевого интерфейса каждые 2 секунды. Работа программы показана на рисунке 3.

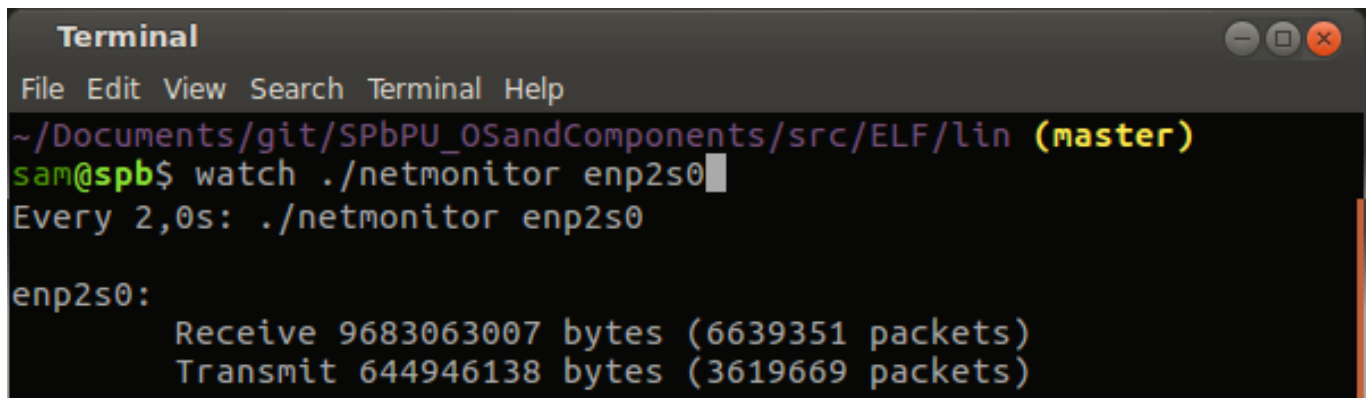
A screenshot of a Linux terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The current directory is `~/Documents/git/SPbPU_OSandComponents/src/ELF/lin (master)`. The user `sam@spb$` has executed the command `watch ./netmonitor enp2s0`. The output shows the command being run every 2 seconds: `Every 2,0s: ./netmonitor enp2s0`. The output of the program is displayed as follows:
`enp2s0:`
`Receive 9683063007 bytes (6639351 packets)`
`Transmit 644946138 bytes (3619669 packets)`

Рис. 3: Исполнение программы

1.6 Динамические библиотеки .so

Библиотека - это набор скомпонованных особым образом объектных файлов. Библиотеки подключаются к основной программе во время линковки. По способу компоновки библиотеки подразделяют на архивы (статические библиотеки, *static libraries*) и совместно используемые (динамические библиотеки, *shared libraries*). В Linux, кроме того, есть механизмы динамической подгрузки библиотек. Суть динамической подгрузки состоит в том, что запущенная программа может по собственному усмотрению подключить к себе какую-либо библиотеку. Благодаря этой возможности создаются программы с подключаемыми плагинами, такие как XMMS.

Статическая библиотека - это просто архив объектных файлов, который подключается к программе во время линковки. Эффект такой же, как при компиляции файлов отдельно.

В отличие от статических библиотек, код совместно используемых (динамических) библиотек не включается в бинарник. Вместо этого в бинарник включается только ссылка на библиотеку.

Рассмотрим преимущества и недостатки статических и совместно используемых библиотек. Статические библиотеки делают программу более автономной: программа, скомпонованная со статической библиотекой может запускаться на любом компьютере, не требуя наличия этой библиотеки (она уже "внутри" бинарника). Программа, скомпонованная с динамической библиотекой, требует наличия этой библиотеки на том компьютере, где она запускается, поскольку в бинарнике не код, а ссылка на код библиотеки. Не смотря на такую зависимость, динамические библиотеки обладают двумя существенными преимуществами. Во-первых, бинарник, скомпонованный с совместно используемой библиотекой меньше размером, чем такой же бинарник, с подключенной к нему статической библиотекой (статически скомпонованный бинарник). Во-вторых, любая модернизация динамической

библиотеки, отражается на всех программах, использующих ее. Таким образом, если некоторую библиотеку `foo` используют 10 программ, то исправление какой-нибудь ошибки в `foo` или любое другое улучшение библиотеки автоматически улучшает все программы, которые используют эту библиотеку. Именно поэтому динамические библиотеки называют совместно используемыми. Чтобы применить изменения, внесенные в статическую библиотеку, нужно пересобрать все 10 программ.

В Linux статические библиотеки обычно имеют расширение `.a` (Archive), а совместно используемые библиотеки имеют расширение `.so` (Shared Object). Хранятся библиотеки, как правило, в каталогах `/lib` и `/usr/lib`. В случае иного расположения (относится только к совместно используемым библиотекам), приходится явно указать путь, чтобы программа запустилась[2].

1.7 Резидентное приложение с динамической библиотекой

В динамическую библиотеку вынесена функция, отвечающая за взаимодействие с системой.

При компиляции, отдельно собирается библиотека, и отдельно исполняемый файл.

```
user@host$ g++ -o libparse.so -shared -fPIC -std=c++14 parse.cpp
user@host$ g++ main.cpp -L. -lparse -o netmonitor
```

Теперь простой запуск приложения приведёт к ошибке, т.к. система ожидает наличия файла библиотеки в строго определённом месте.

```
user@host$ ./netmonitor
./netmonitor: error while loading shared libraries: libparse.so: cannot open shared o
user@host$
```

Отсутствие библиотеки можно легко обнаружить при запуске утилиты `ldd` (строка 2, листинг 6).

Листинг 6: Демонстрация работы программы `ldd` для приложения, использующего динамическую библиотеку

```
1  linux-vdso.so.1 => (0x00007ffc495ca000)
2  libparse.so => not found
3  libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0
    x00007fee33a20000)
4  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fee33656000)
5  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fee3334d000)
```

```
6 /lib64/ld-linux-x86-64.so.2 (0x0000561a68f06000)
7 libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0
  x00007fee33137000)
```

Эту проблему можно обойти если явным образом перед запуском программы передать путь к библиотеке через параметры

```
user@host$ LD_LIBRARY_PATH=. ./netmonitor enp2s0
enp2s0:
    Receive 9686554269 bytes (6645253 packets)
    Transmit 645757402 bytes (3625776 packets)
user@host$
```

Из результатов анализа распределения памяти можно сделать вывод о том, что при запуске программы ей выделяется свободное место в памяти, в следствии чего адреса, по которым располагаются точки входа или подключения меняются при повторных запусках, однако, состав, порядок и смещения загружаемых модулей относительно начального адреса в выделенной памяти не изменяется.

2 Процесс загрузки приложений Windows

2.1 Выполнение EXE-модуля

При запуске EXE-файла (сокр. англ. executable — исполнимый) загрузчик операционной системы создает для его процесса виртуальное адресное пространство и проецирует на него исполняемый модуль. Далее загрузчик анализирует раздел импорта и пытается спроецировать все необходимые DLL на адресное пространство процесса.

Поскольку в разделе импорта указано только имя DLL (без пути), загрузчику приходится самому искать ее на дисковых устройствах в компьютере пользователя. Поиск DLL осуществляется в следующей последовательности.

- Каталог, содержащий EXE-файл.
- Текущий каталог процесса.
- Системный каталог Windows
- Основной каталог Windows
- Каталоги, указанные в переменной окружения PATH.

Проецируя DLL-модули на адресное пространство, загрузчик проверяет в каждом из них раздел импорта. Если у DLL есть раздел импорта (что обычно бывает), загрузчик проецирует следующий DLL-модуль. При этом загрузчик ведет учет загружаемых DLL и проецирует их только один раз, даже если загрузки этих DLL требуют и другие модули.

Если найти файл DLL не удастся, загрузчик выводит сообщение об ошибке.

Найдя и спроецировав на адресное пространство процесса все необходимые DLL-модули, загрузчик настраивает ссылки на импортируемые идентификаторы. Для этого он вновь просматривает разделы импорта в каждом модуле, проверяя наличие указанного идентификатора в соответствующей DLL.

Если идентификатор не найден, то это заканчивается выводом сообщения об ошибке, если же идентификатор найден, загрузчик отыскивает его RVA и прибавляет к виртуальному адресу, по которому данная DLL размещена в адресном пространстве процесса, а затем сохраняет полученный виртуальный адрес в разделе импорта EXE-модуля. И с этого момента ссылка в коде на импортируемый идентификатор приводит к выборке его адреса из раздела импорта вызывающего модуля, открывая таким образом доступ к импортируемой переменной, функции или функции-члену C++ класса. После этого динамические связи установлены, первичный поток процесса начал выполняться, и приложение начинает

работать[4].

Загрузка всех DLL и настройка ссылок занимает какое-то время, но, поскольку такие операции выполняются лишь при запуске процесса, на производительности приложения это не сказывается. Тем не менее, для многих программ подобная задержка при инициализации неприемлема. Чтобы сократить время загрузки приложения, можно модифицировать базовые адреса EXE- и DLL-модулей и провести их (модулей) связывание.

2.2 Динамически подключаемые библиотеки

Динамические библиотеки (dynamic-link libraries, DLL) – краеугольный камень операционной системы Windows, начиная с самой первой ее версии. В DLL содержатся все функции Windows API. Три самые важные DLL:

- Kernel32.dll – управление памятью, процессами и потоками
- User32.dll – поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений
- GDI32.dll – графика и вывод текста.

В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll – стандартные диалоговые окна (вроде File Open и File Save), а ComCtl32.dll поддерживает стандартные элементы управления.

Вот основные причины, по которым инструмент DLL получил такую популярность:

- **Расширение функциональности приложения.** DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний.
- **Возможность использования разных языков программирования.** У разработчика есть выбор, на каком языке писать ту или иную часть приложения. Пользовательский интерфейс приложения можно создавать на Microsoft Visual Basic, а прикладную реализовать на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Коболе, Фортране и др.
- **Более простое управление проектом.** Если в процессе разработки программного

продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов.

- **Экономия памяти.** Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример – DLL-версия библиотеки C/C++. Эту библиотеку используют многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как `sprintf`, `strcpy`, `malloc` и др., будет многократно дублироваться в памяти. Но если они komponуются с DLL-версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.
- **Разделение ресурсов.** DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам.
- **Упрощение локализации.** DLL нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса
- **Решение проблем, связанных с особенностями различных платформ.** В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Windows пользователя не поддерживает эти функции, ему не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной DLL, можно будет загрузить программу даже в более ранних версиях Windows.
- **Реализация специфических возможностей.** Определенная функциональность в Windows доступна только при использовании DLL. Например, отдельные виды ловушек (устанавливаемых вызовом `SetWindowsHookEx` и `SetWinEventHook`) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания COM-объектов, существование которых допустимо только в DLL. Это же относится и к загружаемым Web-браузером ActiveX-элементам, позволяющим создавать Web-страницы с более богатой функциональностью.

Зачастую создать DLL проще, чем приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL

обычно нет кода, предназначенного для обработки циклов выборки сообщений или создания окон. DLL представляет собой набор модулей исходного кода, в каждом из которых содержится определенное число функций, вызываемых приложением (исполняемым файлом) или другими DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, нужно указывать компоновщику ключ /DLL. Тогда компоновщик записывает в конечный файл информацию, по которой загрузчик операционной системы определяет, что данный файл – DLL, а не приложение

Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счёт неявного связывания при загрузке, либо за счет явного – в период выполнения.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто дополнительные код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL ничем не владеет.

Например, если DLL-функция вызывает VirtualAlloc, резервируется регион в адресном пространстве того процесса, которому принадлежит поток, обратившийся к DLL-функции. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион зарезервирован DLL-функций. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет VirtualFree или завершится сам процесс.

2.3 Реализация резидентного приложения

Как и в случае с Linux, приложение отображает количество переданной и полученной информации сетевым интерфейсом. Тут интересно, что используемую в процессе функцию мы получаем непосредственно во время работы (стр. 20 - 22, листинг 7).

Листинг 7: Утилита сбора информации о трафике на сетевых интерфейсах (src/ELF/win/main.cpp)

```
1 #include <iostream>
2 #include <windows.h>
3 #include <iphlpapi.h>
```



```

4
5 bool GetIfTable(PMIB_IFTABLE *m_pTable);
6
7 int main(int argc, char *argv[]) {
8     PMIB_IFTABLE m_pTable = NULL;
9
10    if (GetIfTable(&m_pTable) == false) {
11        return 1;
12    }
13
14    // Обход списка сетевых интерфейсов
15    for (UINT i = 0; i < m_pTable->dwNumEntries; i++) {
16        MIB_IFROW Row = m_pTable->table[i];
17        char szDescr[MAXLEN_IFDESCR];
18        memcpy(szDescr, Row.bDescr, Row.dwDescrLen);
19        szDescr[Row.dwDescrLen] = 0;
20
21        // Вывод собранной информации
22        std::cout << szDescr << ":" << std::endl;
23        std::cout << "\tReceived: " << Row.dwInOctets
24                << ", Sent: " << Row.dwOutOctets << std::endl;
25        std::cout << std::endl;
26    }
27
28    // Завершение работы
29    delete (m_pTable);
30    char a = getchar();
31    return 0;
32 }
33
34 bool GetIfTable(PMIB_IFTABLE *m_pTable) {
35     // Тип указателя на функцию GetIfTable
36     typedef DWORD(_stdcall * TGetIfTable)(
37         MIB_IFTABLE * pIfTable, // Буфер таблицы интерфейсов
38         ULONG * pdwSize,        // Размер буфера
39         BOOL bOrder);           // Сортировать таблицу?
40

```

```

41 // Пытаемся подгрузить iphlapi.dll
42 HINSTANCE iphlapi;
43 iphlapi = LoadLibrary(L"iphlpapi.dll");
44 if (!iphlpapi) {
45     std::cerr << "iphlpapi.dll not supported" << std::endl;
46     return false;
47 }
48
49 // Получаем адрес функции
50 TGetIfTable pGetIfTable;
51 pGetIfTable = (TGetIfTable)GetProcAddress(iphlpapi, "GetIfTable");
52
53 // Получили требуемый размер буфера
54 DWORD m_dwAdapters = 0;
55 pGetIfTable(*m_pTable, &m_dwAdapters, TRUE);
56
57 *m_pTable = new MIB_IFTABLE[m_dwAdapters];
58 if (pGetIfTable(*m_pTable, &m_dwAdapters, TRUE) != ERROR_SUCCESS)
59 {
60     std::cerr << "Error while GetIfTable" << std::endl;
61     delete *m_pTable;
62     return false;
63 }
64
65 return true;
66 }

```

Результат выполнения программы представлен на рисунке 4.

Заполнение таблицы с информацией об интерфейсах (стр. 37, листинг 7) вынесена в отдельную функцию (стр.34, листинг 7). В приложении производится большое количество обращений к различным DLL. Для упрощения дальнейшего анализа вынесем функцию в отдельную динамическую библиотеку (myDllLib) и продолжим анализ.

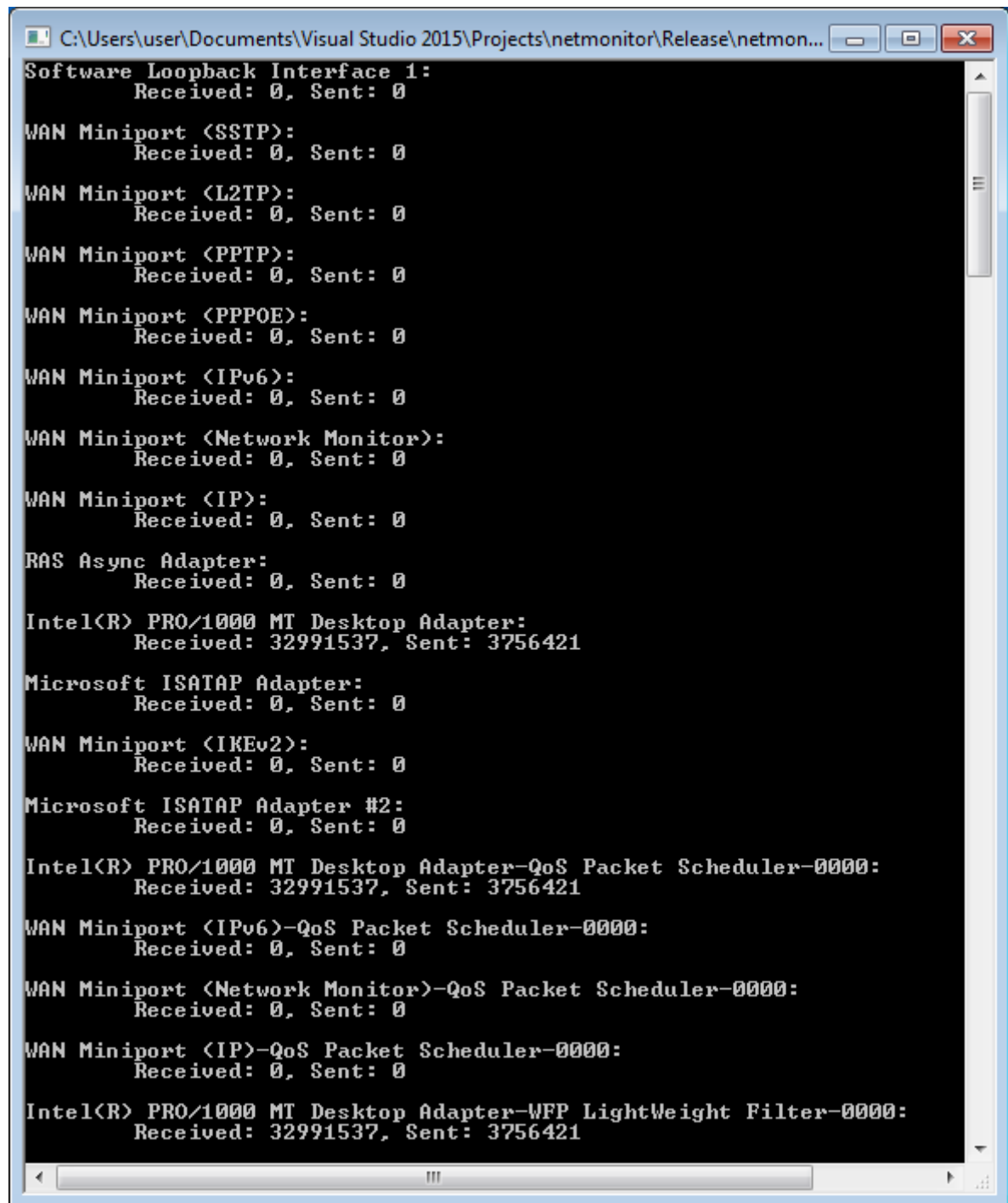


Рис. 4: Исполнение программы в среде Windows

2.4 Анализ исполнения приложения

Утилита API Monitor может декодировать параметры функций и возвращаемые значения, чтобы представить их в понятном формате, а также отобразить точки обращения разработанной динамической myDllLib.dll

#	Time of Day	Thread	Module	API	Return Val
14013	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14014	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14015	12:11:01.054 PM	1	myDllLib.dll		EncodePointe
14016	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14017	12:11:01.054 PM	1	myDllLib.dll		EncodePointe
#	Time of Day	Thread	Module	API	Return Val
14020	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14021	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14022	12:11:01.054 PM	1	myDllLib.dll		EncodePointe
14023	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14024	12:11:01.054 PM	1	myDllLib.dll		EncodePointe
14025	12:11:01.054 PM	1	myDllLib.dll		DecodePointe
14026	12:11:01.054 PM	1	myDllLib.dll		DecodePointe

Отладчик OllyDbg позволяет осуществить разбор пользовательского режима (ring 3). При запуске программы, изначальной точкой входа является функция `_tmainCRTStartup`. Ниже приведена команда вызова данной функции:

CPU Disasm	Address	Hex dump	Command	Comments
	01271BA3	. E8 ADF4FFFF	CALL 01271055;	[__security_init_cookie
	01271BA8	. E8 73FCFFFF	CALL __tmainCRTStartup;	[__tmainCRTStartup

Сама функция `_tmainCRTStartup` находится по адресу 012F1820:

CPU Disasm
Address Hex dump Command Comments
012F1820 /\$ 55 PUSH EBP; INT myDllTest.__tmainCRTStartup(void)

Адрес начала подключения динамической библиотеки находится по адресу 012F1820:

CPU Disasm
Address Hex dump Command Comments
012F1820 /\$ 55 PUSH EBP; INT myDllTest.__tmainCRTStartup(void)
012F1821 . 8BEC MOV EBP,ESP
012F1823 . 6A FE PUSH -2
012F1825 . 68 D06E2F01 PUSH OFFSET 012F6ED0
012F182A . 68 7D102F01 PUSH 012F107D

012F182F	.	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
012F1835	.	50	PUSH EAX
012F1836	.	83C4 E4	ADD ESP,-1C
012F1839	.	53	PUSH EBX
012F183A	.	56	PUSH ESI
012F183B	.	57	PUSH EDI
012F183C	.	A1 24802F01	MOV EAX,DWORD PTR DS:[__security_cookie]
012F1841	.	3145 F8	XOR DWORD PTR SS:[EBP-8],EAX
012F1844	.	33C5	XOR EAX,EBP
012F1846	.	50	PUSH EAX
012F1847	.	8D45 F0	LEA EAX,[EBP-10]
012F184A	.	64:A3 00000000	MOV DWORD PTR FS:[0],EAX
012F1850	.	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
012F1853	.	C745 FC 000000	MOV DWORD PTR SS:[EBP-4],0
012F185A	.	C745 E4 000000	MOV DWORD PTR SS:[EBP-1C],0

Адреса точки входа и подключения библиотек не изменились.

Process Monitor является усовершенствованным инструментом отслеживания для Windows, который в режиме реального времени отображает активность файловой системы, реестра, а также процессов и потоков. Проверим адреса подключения динамической библиотеки при помощи утилиты Process Monitor.

Description:	
Company:	
Name:	myDllTest.exe
Version:	
Path:	C:\Projects\myDllLib\Debug\myDllTest.exe
Command Line:	myDllTest.exe
PID:	3380
Parent PID:	2436
Session ID:	1
User:	sba-PC\sba
Auth ID:	00000000:0001b9ac
Architecture:	32-bit
Virtualized:	False
Integrity:	Обязательная метка\Средний обязательный уровень
Started:	22.03.2016 2:02:47

Ended:	22.03.2016 2:04:46		
Modules:			
myDllTest.exe	0x250000	0x1c000	C:\Projects\myDllLib\Debu
22.03.2016 0:45:34			
MSVCR120D.dll	0x72e10000	0x1bf000	C:\Windows\SysWOW64\MS
Microsoft Corporation	12.00.21005.1	built by: REL	05.10.2013 5:
myDllLib.dll	0x73be0000	0x1b000	C:\Projects\myDllLib\Deb
22.03.2016 0:45:33			
wow64win.dll	0x74b40000	0x5c000	C:\Windows\SYSTEM32\wow6
Microsoft Corporation	6.1.7601.19018	(win7sp1_gdr.150928-1507)	
wow64.dll	0x74ba0000	0x3f000	C:\Windows\SYSTEM32\wow64.d
Microsoft Corporation	6.1.7601.19018		
(win7sp1_gdr.150928-1507)	29.09.2015 6:12:08		
wow64cpu.dll	0x74c10000	0x8000	C:\Windows\SYSTEM32\wow64
Corporation	6.1.7601.19018	(win7sp1_gdr.150928-1507)	
29.09.2015 6:12:09			
KERNELBASE.dll	0x75220000	0x47000	C:\Windows\syswow64\KE
Corporation	6.1.7601.18015	(win7sp1_gdr.121129-1432)	
29.09.2015 6:00:36			
kernel32.dll	0x75350000	0x110000	C:\Windows\syswow64\ker
Corporation	6.1.7601.18015	(win7sp1_gdr.121129-1432)	
29.09.2015 6:00:35			
ntdll.dll	0x77100000	0x1a9000	C:\Windows\SYSTEM32\ntdll.
Microsoft Corporation	6.1.7600.16385		
(win7_rtm.090713-1255)	29.09.2015 6:07:47		
ntdll.dll	0x772e0000	0x180000	C:\Windows\SysWOW64\ntdll.
Microsoft Corporation	6.1.7600.16385		
(win7_rtm.090713-1255)	29.09.2015 5:57:52		

Как видно из вывода, тестовая динамическая библиотека myDllLib.dll была подключена по адресу 0x73be0000 и имеет размер 0x1b000.

Выделение памяти в ОС Windows и Linux схожи: и в том и в другом случае программе выделяется произвольная доступная область памяти, вследствие чего адреса подключения модулей различаются, однако порядок подключения модулей, их смещения относительно выделенной области памяти, а так же виртуальный адрес точки входа остается неизменным.

Заключение

В данной работе были рассмотрены механизмы загрузки исполняемых приложений в операционных системах семейства Windows и Linux.

Современные подходы к разработке больших приложений предполагают использование динамических библиотек, обладающих своими особенностями.

Достоинства:

- экономия памяти за счёт использования одной библиотеки несколькими процессами;
- разработка различных модулей на различных языках;
- возможность исправления ошибок (достаточно заменить файл библиотеки и перезапустить работающие программы).

Недостатки:

- возможность нарушения API (при внесении изменений в библиотеку существующие программы могут перестать работать);
- конфликт версий динамических библиотек (разные программы могут ожидать разные версии библиотек);
- доступность одинаковых функций по одинаковым адресам в разных процессах (упрощает эксплуатацию уязвимостей).

Главной особенностью динамических библиотек является ускорение процесса разработки и предоставление хорошо протестированных решений, что является важнейшими задачами в индустрии.

Список литературы

- [1] Лав Р. Linux. Системное программирование. 2-е изд. – СПб.: Питер, 2014 – 448 стр.
- [2] Рэймонд Э.С.. Искусство программирования для Unix. – М.: Вильямс, 2005 – 544 стр.
- [3] Касперски К. Секреты поваров компьютерной кухни или ПК: решение проблем – М.: BHV, 2003 – 560 стр.
- [4] Харт Дж. Системное программирование в среде Windows. – М.: 2005