Создание простого приложения для операционных систем UNIX-семейства.

Краткий справочный материал

Unix-подобная операционная система

Unix-подобная операционная система - операционная система, которая образовалась под влиянием **Unix**. Термин включает свободные/открытые операционные системы, образованные от **Unix** компании Bell Labs или эмулирующие его возможности, коммерческие и запатентованные разработки, а также версии, основанные на исходном коде **Unix**.

GNU General Public License

GNU General Public License (переводят как Универсальная общественная лицензия **GNU**, Универсальная общедоступная лицензия **GNU** или Открытое лицензионное соглашение **GNU**) — лицензия на свободное программное обеспечение, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют **GNU GPL** или даже просто **GPL**.

Дополнительная информация:

- https://www.gnu.org/licenses/gpl-3.0.html

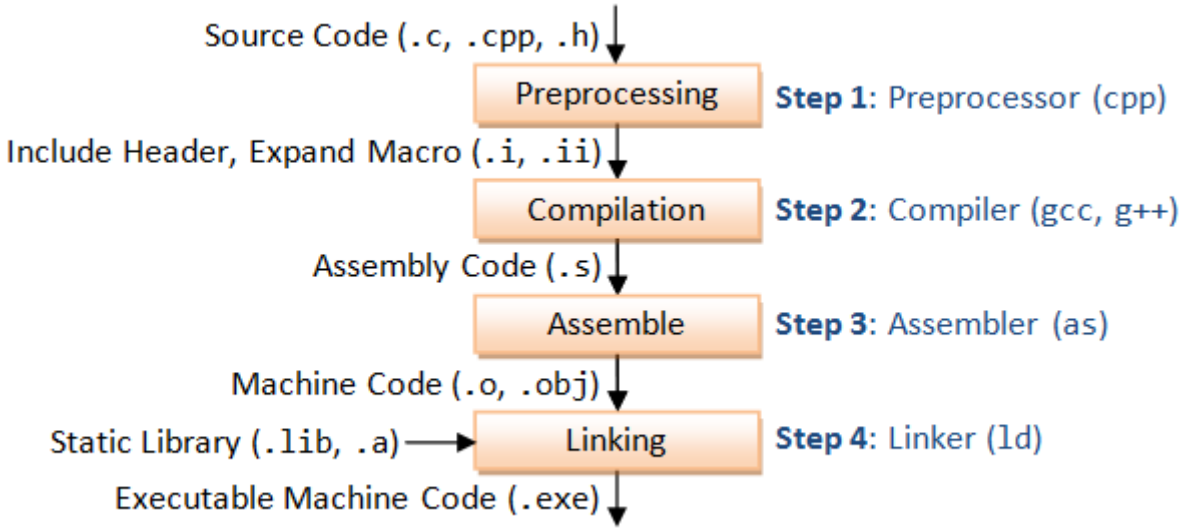
Компилятор **gcc**

Си (от лат. буквы **C**, англ. языка) — компилируемый статически типизированный язык программирования общего назначения.Данный язык доступен на самых различных платформах. Несмотря на свою низкоуровневую природу, язык ориентирован на переносимость. Программы, соответствующие стандарту языка, могут компилироваться под различные архитектуры компьютеров.

GNU Compiler Collection (обычно используется сокращение **GCC**) — набор компиляторов для различных языков программирования, разработанный в рамках проекта **GNU**. **GCC** является свободным программным обеспечением, распространяется в том числе фондом свободного программного обеспечения (**FSF**) на условиях **GNU GPL** и **GNU LGPL** и является ключевым компонентом **GNU toolchain**. Он используется как стандартный компилятор для свободных **UNIX**-подобных операционных систем.

Стадии сборки приложений на C

Классический сценарий сборки кода на C включает четыре этапа.



Препроцессинг

Этап подставляет `include`-файлы, генерирует код с помощью макросов, заменяет `define`-константы на их значения.

Посмотреть результат препроцессинга можно через

```
gcc -E main.c
```

Компиляция

Обработка исходного кода (уже без директив препроцессора) и преобразование его в команды ассемблера для целевой платформы (например x86).

```
gcc -S main.c
```

Результат записывается в файл `main.s`.

Ассемблирование

Преобразование кода на языке ассемблера в бинарный формат — в объектный файл.

```
gcc -c main.s
```

Результат записывается в файл `main.o`.

Компоновка (linking)

Объектный код, сгенерированный ассемблером, компонуется с другими объектным кодом, в том числе с библиотеками, для получения исполняемого файла.

```
gcc main.o -o program
```

Дополнительная информация:

- https://acm.bsu.by/wiki/C2017/Сборка_программ_на_C
- https://web.archive.org/web/20181024175400/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf
- https://habr.com/ru/articles/657209/
- https://runebook.dev/ru/docs/gcc/-index-
- https://gcc.gnu.org/

Набор библиотек **gcc-multilib**

`gcc-multilib` представляет собой набор библиотек и заголовочных файлов, необходимых для поддержки сборки и выполнения 32-битных приложений на 64-битных системах. По умолчанию **Linux** ОС поставляется с 64-битной версией `gcc`, которая может компилировать только 64-битный код. Однако при разработке программного обеспечения часто возникает необходимость в сборке и тестировании 32-битных версий программ, особенно если ваше приложение предназначено для работы на разных архитектурах или операционных системах.

Дополнительная информация:

- https://qaa-engineer.ru/chto-imenno-oznachaet-gcc-multilib-v-ubuntu/

Утилита **file**

Команда `file` предназначена для задействования одноименной утилиты, осуществляющей определение типов переданных элементов файловой системы (файлов, директорий, ссылок, именованных каналов и сокетов). Данная утилита исследует содержимое файлов, а не ограничивается проверкой их расширений.

Дополнительная информация:

- https://linux-faq.ru/page/komanda-file
- https://losst.pro/komanda-file-v-linux
- https://linuxcommandlibrary.com/man/file

Утилита **make**

`make` - утилита предназначенная для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем `Makefile`, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил, которые в свою очередь описываются:

1. целями (то, что данное правило делает);
2. реквизитами (то, что необходимо для выполнения правила и получения целей);

3. командами (выполняющими данные преобразования).

В общем виде синтаксис makefile можно представить так:

```
# Индентация осуществляется исключительно при помощи символов табуляции,
# каждой команде должен предшествовать отступ
<цели>: <реквизиты>
    <команда #1>
    ...
    <команда #n>
```

Дополнительная информация:

- https://habr.com/ru/articles/211751/
- https://www.opennet.ru/docs/RUS/make_compile/
- https://www.gnu.org/software/make/manual/

Отладчик `gdb`

GNU Debugger (`gdb`) — переносимый отладчик проекта **GNU**, который работает на многих **UNIX**-подобных системах и умеет производить отладку многих языков программирования, включая Си, C++, Free Pascal, FreeBASIC, Ada, Фортран и Rust. **GDB** — свободное программное обеспечение, распространяемое по лицензии GPL.

Точка входа (англ. **Entry Point (EP)** — точка входа) — адрес в оперативной памяти (в виртуальном адресном пространстве процесса), с которого начинается выполнение программы. Другими словами — адрес, по которому хранится первая команда программы.

Дополнительная информация:

- https://www.sourceware.org/gdb/
- https://www.opennet.ru/docs/RUS/gdbint/
- https://www.opennet.ru/docs/RUS/gdb/
- https://stackru.com/questions/19688412/kak-najti-tochku-vhoda-osnovnoj-funktsii-ispolnyaemogo-fajla-elf-bez-kakoj-libo-simvolicheskoj-informatsii
- http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

Ассемблер `nasm`

Язык ассемблера (англ. assembly language) — представление команд процессора в виде, доступном для чтения человеком. Язык ассемблера считается языком программирования низкого уровня, в противовес высокоуровневым языкам, не привязанным к конкретной реализации вычислительной системы. Программы, написанные на языке ассемблера однозначным образом переводятся в инструкции конкретного процессора и в большинстве случаев не могут быть перенесены без значительных изменений для запуска на машине с другой системой команд. Ассемблером называется программа, преобразующая код на языке ассемблера в машинный код; программа, выполняющая обратную задачу, называется **дизассемблером**.

NASM (Netwide Assembler) — свободный (**LGPL** и лицензия **BSD**) ассемблер для архитектуры **Intel x86**. Используется для написания 16-, 32- и 64-разрядных программ.

В NASM используется Intel-синтаксис записи инструкций. Предложение языка ассемблера NASM (строка программы) может состоять из следующих элементов:

```
Метка Инструкция Операнды Комментарий
```

Операнды разделяются между собой запятой. Перед строкой и после инструкции можно использовать любое количество пробельных символов. Комментарий начинается с точки с запятой, а концом комментария считается конец строки. В качестве инструкции может использоваться команда или псевдокоманда (директива компилятора).

Компиляция программ в NASM состоит из двух этапов. Первый — ассемблирование, второй — компоновка. На этапе ассемблирования создаётся объектный код. В нём содержится машинный код программы и данные, в соответствии с исходным кодом, но идентификаторы (переменные, символы) пока не привязаны к адресам памяти. На этапе компоновки из одного или нескольких объектных модулей создаётся исполняемый файл (программа). Операция компоновки связывает идентификаторы, определённые в основной программе, с идентификаторами, определёнными в остальных модулях, после чего всем идентификаторам даются окончательные адреса памяти или обеспечивается их динамическое выделение.

Для компоновки объектных файлов можно использовать свободный бесплатно распространяемый компоновщик `ld`, который есть в любой версии Linux.

Для ассемблирования файла нужно ввести следующую команду:

```
nasm -f format filename -o output
```

Дополнительная информация:

- https://eax.me/assembler-basics/
- http://www.stolyarov.info/books/asm_unix
- http://www.nasm.us/

Программное обеспечение, необходимое для выполнения лабораторной работы

1. Программный продукт виртуализации для операционных систем VirtualBox. https://www.virtualbox.org/. (либо его аналог)
2. Kali Linux - GNU/Linux-LiveCD. https://www.kali.org/ (самостоятельно или для создания виртуального жесткого диска).
3. Виртуальный жесткий диск (VDI, VHD, VMDK, HDD, QSOW, QED) с установленной операционной системой Linux (Astra Linux, Debian, Ubuntu, Mint, Kali и т.д.)
4. Установленные пакеты : `make`, `gcc`, `gcc-multilib`, `nasm`, `gdb`

Задание 0. Подготовка виртуальной машины.

1. Создайте новую виртуальную машину (можете использовать уже существующую с установленной операционной системой **Linux** (Astra Linux, Debian, Ubuntu, Mint, Kali и т.д.).
2. Установите операционную систему.
3. Установите пакеты : `make`, `gcc`, `gcc-multilib`, `nasm` либо проверьте их наличие в системе.
4. Проверьте наличие в системе текстового редактора (произвольно: `nano`, `mousepad`, `gedit` и т.д.).

Задание 1. Создание простого 32-битного приложения на языке программирования **C**.

1. Запустите виртуальную машину с установленной операционной системой **Linux**.

2. В рабочей директории создайте каталог `c`. Перейдите в него.

3. С использованием произвольного доступного в вашей **Linux** операционной системе текстововго редактора, создайте файл `main.c` со следующим содержимым:

```
#include "stdio.h"

int main()
{
  printf("Hello World!\n");
   return 0;
}
```

4. В строке `printf("Hello World!\n")` замените строку `Hello World!` на строку `"Ivanov 10 32bit"` (ваша фамилия транслитом и номер в списке группы).

5. Для подачи команд используйте утилиты командной строки вашей версии **Linux**. (Например для Kali Linux: `QTerminal`, `xterm` и т.д.)

6. Откомпилируйте исходный код `main.c` в исполняемый файл 32-битной системы выполнения **Linux** путем подачи команды:

```
gcc -m32 -o hello32 main.c
```

7. Распечатайте содержимое рабочей директории (команда `ls`), убедитесь, что в списке файлов появился файл `hello32`.

8. Выполните команду `file` для получения информации и файле:

```
file hello32
```

9. Сохраните информацию о файле `hello32` в текстовый файл `hello32_info.txt` (с применением перенаправления вывода) путем подачи команды:

```
file hello32 > hello32_info.txt
```

10. Запустите файл на ислолнение путем подачи команды:

```
./hello32
```

11. Выясните размер исполняемого файла `hello32` путем подачи команды:

```
ls -lh
```

Задание 2. Создание простого 64-битного приложения на языке программирования **C**.

1. Внесите изменения в файл `main.c`: в строке `printf("Ivanov 10 32bit\n")` замените строку `Ivanov 10 32bit` на строку `Ivanov 10 64bit` (ваша фамилия транслитом и номер в списке группы). Сохраните файл.
2. Откомпилируйте исходный код `main.c` в исполняемый файл 64-битной системы выполнения **Linux** путем подачи команды:

```
gcc -m64 -o hello64 main.c
```

3. Распечатайте содержимое рабочей директории (команда `ls`), убедитесь, что в списке файлов появился файл `hello64`.
4. Выполните команду `file` для получения информации и файле:

```
file hello64
```

5. Сохраните информацию о файле `hello64` в текстовый файл `hello64_info.txt` (с применением перенаправления вывода) путем подачи команды:

```
file hello64 > hello64_info.txt
```

6. Запустите файл на ислолнение путем подачи команды:

```
./hello64
```

7. Выясните размер исполняемого файла `hello64` путем подачи команды:

```
ls -lh
```

8. Вычислите разницу в размере исполняемого файла для 32-битной системы выполнения и 64-битной.

Задание 3. Применение утилиты **make** для сборки и создания исполняемых модулей. Часть 1.

1. Ознакомьтесь с базовой информацией об утилите `make`.

2. В рабочей директории, в каталоге `c`, с использованием произвольного доступного в вашей **Linux** операционной системе текстововго редактора, создайте файл `Makefile` со следующим содержимым:

```
hello: main.c
    gcc -o hello main.c

hello32: main.c
    gcc -m32 -o hello32 main.c

hello64: main.c
    gcc -m64 -o hello64 main.c

clean:
    rm -rf hello hello32 hello64

edit:
    nano main.c
```

3. В рабочей директории, в каталоге `c`, в терминале выполните команду, которая удалит все исполняемые модули: `make clean`

4. С применением команды `ls` распечатайте содержимое рабочего каталога и убедитесь, что все полученные ранее в результате компиляции исполняемые файлы удалены.

5. При помощи утилиты `make` получите (соберите) 32-битное приложение при помощи подачи в терминале команды:

```
make hello32
```

6. Убедитесь, что компиляция прошла успешно. запустите полученный исполняемый файл (версия для 32-битной среды выполнения) в терминале.

7. При помощи утилиты `make` получите (соберите) 64-битное приложение.

8. Убедитесь, что компиляция прошла успешно. запустите полученный исполняемый файл (версия для 64-битной среды выполнения) в терминале.

9. С применением утилиты `file` распечатайте информацию о получившихся в результате применения утилиты `make` исполняемых файлах.

10. В рабочей директории, в каталоге `c`, в терминале выполните команду:

```
make edit
```

Опишите результат выполнения команды.

Задание 4. Создание простого 64-битного приложения на языке программирования **Ассемблер**.

1. В рабочей директории создайте каталог `asm`. Перейдите в него.

2. С использованием произвольного доступного в вашей **Linux** операционной системе текстововго редактора, создайте файл `hello64.asm` со следующим содержимым (комментарии, т.е. текст после символа ";" включительно можно не вставлять):

```
section .data
 msg     db "Hello World!", 10     ; db: data byte, 10: ASCII newline
section .text
 global _start
_start:
 mov     rax, 1                     ; write
 mov     rdi, 1                     ; to stdout
 mov     rsi, msg                   ; starting at msg
 mov     rdx, 13                    ; for len bytes
 syscall
 mov     rax, 60                   ; exit
 mov     rdi, 0                     ; with success
 syscall                           ;_
```

3. В файле `hello64.asm` замените строку `Hello World!` на строку `"Ivanov 15 64 bit"` (ваша фамилия транслитом и номер в списке группы).

4. Для полученииния исполняемого модуля в терминале выполните следующие команды:

```
nasm -f elf64 hello.asm -o hello64.o
ld hello64.o -o hello64
```

5. Убедитесь, что компиляция и сборка прошли успешно. запустите полученный исполняемый файл в терминале.

6. С применением утилиты `file` распечатайте информацию об исполняемом файле `hello64`.
7. Сравните размеры исполняемых файлов (64-битные версии исполняемых файлов) полученных в этом задании и файлами, которые были получены по результатам выполнения предыдущих заданий.

Задание 5. Создание простого 32-битного приложения на языке программирования **Ассемблер**.

1. В директории `asm`, с использованием произвольного доступного в вашей **Linux** операционной системе текстововго редактора, создайте файл `hello32.asm` со следующим содержимым:

```
section .text
global _start

_start:
 mov ebx, 0x1
 mov ecx, hello
 mov edx, helloLen
 mov eax, 0x4
 int 0x80

 xor ebx, ebx
 mov eax, 0x1
 int 0x80

section .data
 hello db "Hello World!", 0xa
 helloLen equ $-hello
```

2. В файле `hello32.asm` замените строку `Hello World!` на строку `"Ivanov 15 32 bit"` (ваша фамилия транслитом и номер в списке группы).
3. Для полученииния исполняемого модуля в терминале выполните следующие команды:

```
nasm -f elf32 hello32.asm -o hello32.o
ld -m elf_i386 hello32.o -o hello32
```

4. Убедитесь, что компиляция и сборка прошли успешно. запустите полученный исполняемый файл в терминале.
5. С применением утилиты `file` распечатайте информацию об исполняемом файле `hello32`.
6. Сравните размеры исполняемых файлов (32-битные версии исполняемых файлов) полученных в этом задании и файлами, которые были получены по результатам выполнения предыдущих заданий.

Задание 6. Применение утилиты **make** для сборки и создания исполняемых модулей. Часть 2.

1. Разработайте `Makefile` для сборки 32- и 64-битных приложений из исходного кода на языке **Ассемблер**.
2. Протестируйте все режимы сборки (32- и 64-битные исполняемые файлы).

Задание 7. Определение точки входа в 32-битное приложение.

1. Перейдите в подкаталог `asm` рабочего каталога и убедитесь в наличии исполняемого файла `hello32` (в случае его отсутствия произведите компиляцию и сборку указанногго файла).
2. Проверьте наличие установленного отладчика `gdb`, выполнив команду из консоли:

```
gdb
```

3. Убедитесь, что на экране вы видите консоль отладчика:

```
(gdb)
```

4. Выйдете из отладчика путем подачи команды `q`:

```
(gdb) q
```

5. Запустите отладчик `gdb` из консоли с параметрами:

```
gdb -q -nh hello32
```

6. В запущенном отладчике подайте команду `info file`:

```
(gdb) info file
```

7. Убедитесь, что вы получили подобную информацию (значения адресов могут отличаться) в консоли:

```
Symbols from "/home/kali/asm/hello32".
Local exec file:
     `/home/kali/asm/hello32', file type elf32-i386.
     Entry point: 0x8049000
     0x08049000 - 0x0804901f is .text
     0x0804a000 - 0x0804a00c is .data
```

8. Зафиксируйте адрес точки входа в приложение (например: `Entry point: 0x8049000`).

9. Установите точку останова (`breakpoint`) на значении точки входа в приложение (`Entry point`) (для того, чтобы после запуска приложения в отладчике, его выполнение остановилось на точке входа) путем подачи команды `break *0x8049000` (или сокращенный вариант `b *0x8049000`):

```
(gdb) break *0x8049000
```

10. Убедитесь, что точка останова установлена:

```
Breakpoint 1 at 0x8049000
```

11. Запустите приложение в отладчике путем подачи команды `run`:

```
(gdb) run
```

12. Убедитесь, что вы запустили приложение в отладчике и он вывел сообщение о запуске:

```
Starting program: /home/kali/asm/hello32

Breakpoint 1, 0x08049000 in _start ()
```

13. Дизассемблируйте (переведите машинный код в инструкции ассемблера) первые восемь инструкций программы, начиная с точки останова (адрес точки останова содержится в счетчике команд, т.е. регистре `eip` (для 32-битных приложений) путем подачи команды `x/8i $eip`:

```
(gdb) x/8i $eip
```

14. Убедитесь, что результат дизассемблирования будет (примерно) таким, как представлено здесь:

```
=> 0x8049000 <_start>:          mov    $0x1,%ebx
   0x8049005 <_start+5>:        mov    $0x804a000,%ecx
   0x804900a <_start+10>:       mov    $0xc,%edx
   0x804900f <_start+15>:       mov    $0x4,%eax
   0x8049014 <_start+20>:       int    $0x80
   0x8049016 <_start+22>:       xor    %ebx,%ebx
   0x8049018 <_start+24>:       mov    $0x1,%eax
   0x804901d <_start+29>:       int    $0x80
```

15. Сравните исходный код с дизассембированным, сделайте выводы по отличиям.

Задание 8. Определение точки входа в 64-битное приложение.

1. Перейдите в подкаталог `asm` рабочего каталога и убедитесь в наличии исполняемого файла `hello64` (в случае его отсутствия произведите компиляцию и сборку указанногго файла).
2. Проделайте все шаги задания 7 для 64-битного приложения hello64 (с учетом того, что название счетчика команд в 64-битном режиме `rip`)
3. Зафиксируйте адрес точки входа в приложение и дизассемблированный код.