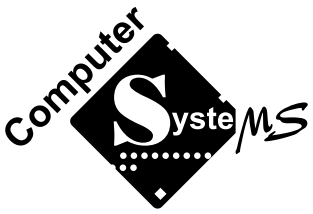
МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Кафедра электронных вычислительных машин и систем

Д. А. Костюк, Г. А. Четверкина

Программирование на ассемблере в GNU/Linux

Методическое пособие
для студентов специальностей 1-40 01 02 и 1-36 04 02

Брест

2013

УДК 004.431.4
ББК 32.844
Д36

Рецензенты:

доцент кафедры радиофизики Львовского национального университета им. И. Франко, кандидат технических наук, доцент Злобин Григорий Григорьевич;

доцент кафедры электронных вычислительных машин Белорусского государственного университета информатики и радиоэлектроники ...

Д. А. Костюк, Г. А. Четверкина
Д36 Программирование на ассемблере в GNU/Linux: методическое пособие. — Брест: изд-во БрГТУ, 2013. — 68 с.

ISBN 978-985-493-184-5

Методическое пособие содержит краткий практический курс программирования на ассемблере для операционной системы GNU/Linux, включающий изложение базовых принципов построения ассемблерных программ, основы использования необходимых инструментальных средств, а также темы, касающиеся использования наиболее распространенных машинных команд архитектуры x86 и системных вызовов ядра Linux. Методическое пособие предназначено для студентов специально- стей 1-40 01 02 «Вычислительные машины, системы и сети» и 1-36 04 02 «Промышленная электроника».

УДК 621.396
ББК 32.844

ISBN 978-985-493-184-5

Оглавление

Введение

Изучение программирования на ассемблере важно с двух позиций. В первую очередь, знание машинных команд актуально при разработке драйверов устройств и других аппаратно-зависимых частей операционных систем, при создании компиляторов и интерпретаторов языков программирования, при решении задач, требующих жесткой оптимизации критических участков кода. Кроме того, навыки программирования на ассемблере важны для понимания внутренней структуры и принципов работы микропроцессорной техники. Составление и отладка ассемблерных программ является единственным способом практического знакомства с внутренним устройством и функционированием процессоров. Знание основ ассемблера увеличивает способность программиста генерировать качественный и эффективный код на языках высокого уровня, избавляет программные продукты от ряда «узких мест» производительности, уменьшает число потенциальных уязвимостей в коде.

GNU/Linux — современная, динамично развивающаяся операционная система общего назначения, успешно масштабируемая для различных сегментов рынка. Открытый доступ к исходному коду значительно облегчают ее адаптацию для самых различных аппаратных платформ и областей применения. Этим объясняются сильные позиции данной системы на рынке специализированных устройств. Она используется в потребительской электронике, телекоммуникационном оборудовании, медицинских системах, автомобилях, вооружении и др. Аналогичная ситуация наблюдается и на рынке мобильных устройств, с учетом доминирующего положения базирующейся на Linux платформы Android. В ряде стран, включая Евросоюз, выполняются государственные программы, направленные на замещение GNU/Linux и сопутствующей инфраструктурой ПО аналогичных проприетарных продуктов в госсекторе, обоснованные как экономическими преимуществами, так и простотой выполнения аудита кода. Эти же соображения обеспечивают GNU/Linux устойчивую популярность в сегменте серверных операционых систем.

На текущий момент платформа GNU/Linux может быть признана наиболее удобной и для изучения низкоуровневого программирования. С точки зрения программирования на ассемблере Linux характеризуется строго унифицированным интерфейсом системных вызовов, одинаковым для всех функций ядра операционной системы. Изучение облегчают и унаследованные традиции ОС Unix и стандарта POSIX, привносящее много общего со стандартными функциями языка C, хорошо знакомого студентам специальностей информатики и радиоэлектроники.

Озвученные доводы послужили отправной точкой для разработки настоящего методического пособия.

1 Первая программа на языке ассемблера

1.1 Краткие теоретические сведения

1.1.1 Структура ассемблерной программы

Каждый язык программирования имеет свои особенности. Язык ассемблера - не исключение, особенно, если программировать на нем на платформе Linux. Рассмотрим пример простой программы на этом языке под эту платформу. Традиционно первая программа выводит приветственное сообщение на экран.

```
    ;
SECTION .data
    hello:     DB 'Hello world!',10 ; 'Hello world!' плюс символ
                                    ; возврата каретки
    helloLen:  EQU $-hello          ; Длина строки 'Hello world!'

SECTION .text               ; Начало секции кода
    GLOBAL _start           ; Метка _start должна быть глобальной,
                            ; чтобы линкер смог её найти и сделать
                            ; точкой входа в программу.
_start:
    mov eax,4               ; Системный вызов для записи (sys_write)
    mov ebx,1               ; Описатель файла 1 - стандартный вывод
    mov ecx,hello           ; Адрес строки hello в ecx
    mov edx,helloLen        ; helloLen - константа, а не переменная,
                            ; потому нет необходимости использовать
                            ; mov edx,[helloLen] для получения
                            ; действительного значения

    int 80h                 ; Вызов ядра

    mov eax,1               ; Системный вызов для выхода (sys_exit)
    mov ebx,0               ; Выход с кодом возврата 0 (без ошибок)
    int 80h                 ; Вызов ядра
```

В отличие от многих современных высокоуровневых языков программирования, в ассемблерной программе каждая команда располагается на *отдельной строке*. Нельзя разместить несколько команд на одной строке. Не принято также разбивать одну команду на несколько строк.

Синтаксис ассемблера NASM является *регистрочувствительным*. Т.е. есть разница между большими и малыми буквами.

Команда может быть *директивой* — указанием транслятору, которое выполняется в процессе превращения программы в машинный код. Многие директивы начинаются с точки. Для удобства чтения программы они обычно пишутся БОЛЬШИМИ БУКВАМИ. Кроме директив еще бывают *инструкции* — команды процессору. Именно они и будут составлять машинный код программы.

Нужно отметить, что понятие «машинного кода» очень условно. Часто оно обозначает просто содержимое выполняемого файла, хра-

нящего кроме собственно машинных команд еще и данные (в нашем случае — текст выводимого сообщения «Hello world»).

1.1.2 Особенности создания ассемблерной программы

На платформе Linux язык ассемблера является самым низкоуровневым языком программирования. Т.е. он больше любых других приближен к архитектуре ЭВМ и ее аппаратным возможностям, позволяет получить к ним более полный доступ, нежели в языках выского уровня, наподобие C/C++, Perl, Python и пр. Заметим, что получить полный доступ к ресурсам компьютера в современных архитектурах нельзя, самым низким уровнем работы программы является обращение напрямую к ядру ОС. Именно на этом уровне и работают программы, написанные на ассемблере в Linux. Но, в отличие от языков высокого уровня (ЯВУ), ассемблерная программа содержит только тот код, который *ввел программист*, и конечно же вся ответственность за логичность кода *полностью* лежит на плечах *программиста*.

Простой пример. Обычно подпрограммы заканчиваются командой возврата. Если в ЯВУ ее не задать явно, транслятор все равно добавит ее в конец подпрограммы. Ассемблерная подпрограмма без команды возврата *не вернется* в точку вызова, а будет выполнять код, следующий за подпрограммой, как будто он является ее продолжением.

Еще пример. Можно попробовать «выполнить» данные вместо кода. Часто это лишено смысла. Но если программист это сделает, транслятор не выдаст никаких сообщений об ошибке. Язык ассемблера позволяет делать все! Вопрос состоит лишь в том, какие усилия придется приложить, чтобы реализовать идею на этом языке. Тут меньше ограничений, чем в ЯВУ, но, в то же время, и удобства и простоты создания программ тоже меньше.

Эти особенности приводят к тому, что ассемблерные программы часто «подвешивают» компьютер, особенно у начинающих программистов. Выделим разновидности «зависания» по способу борьбы с ним.

- Простое — для выхода из него достаточно нажать Ctrl+C (сначала нажимается клавиша Ctrl, и дальше нужно, *не отпуская* ее, нажать вторую клавишу — C; затем клавиши отпускаются в любом порядке). Программа при этом аварийно завершается выходом в ОС.
- Мягкое — кажется, что машина никак не реагирует на клавиатуру и безнадежно зависла. В любом случае, ядро системы при этом продолжает работать и позволяет использовать базовые функции для сохранения целостности данных. Этими функциями мож-

но управлять при помощи т. н. Magic Keys (см. описание SysRq Keys).
- Жесткое — если зависло ядро ОС. Это может случиться в случае использования тестового ядра, находящегося в разработке, или при неправильной ручной сборке ядра, или при попытке использовать недокументированные особенности аппаратного обеспечения. В этом случае поможет аппаратный сброс при помощи кнопки «Reset», расположенной на передней панели системного блока.

Важно помнить, что в 90% случаев зависание является простым. Чаще всего не хватает аппаратных возможностей компьютера для быстрой обработки данных и необходимо просто подождать или нажать Ctrl+C.

1.1.3 Процесс обработки программы на языке ассемблера

Из-за специфики программирования, а также по традиции, для создания программ на языке ассемблера обычно пользуются утилитами командной строки (хотя поддержка ассемблера и есть в некоторых универсальных интегрированных средах). Весь процесс технического создания ассемблерной программы можно разбить на 4 шага (исключены этапы создания алгоритма, выбора структур данных и т.д.).

1. Набор программы в текстовом редакторе и сохранение ее в отдельном файле. Каждый файл имеет имя и тип, называемый иногда расширением. Тип в основном используется для определения назначения файла. Например, программа на C имеет тип `c`, на Pascal — `pas`, на языке ассемблера — `asm`.
2. Обработка текста программы транслятором. На этом этапе текст превращается в машинный код, называемый объектным. Кроме того есть возможность получить листинг программы, содержащий кроме текста программы различную дополнительную информацию и таблицы, созданные транслятором. Тип объектного файла — `o`, файла листинга — `lst`. Этот этап называется *трансляцией*.
3. Обработка полученного объектного кода компоновщиком. Тут программа «привязывается» к конкретным условиям выполнения на ЭВМ. Полученный машинный код называется выполняемым. Кроме того, обычно получается карта загрузки программы в ОЗУ. Выполняемый файл обычно не имеет расширения в отличие от программ ОС семейства DOS и Windows, карта загрузки — `map`. Этот этап называется *компоновкой* или *линковкой*.

4. Запуск программы. Если программа работает не совсем корректно, перед этим может присутствовать этап *отладки* программы при помощи специальной программы — отладчика. При нахождении ошибки приходится проводить коррекцию программы, возвращаясь к шагу 1.

Таким образом, процесс создания ассемблерной программы можно изобразить в виде следующей схемы. Конечной целью, напомним, является работоспособный выполняемый файл `hello` (см. рис. 1.1).

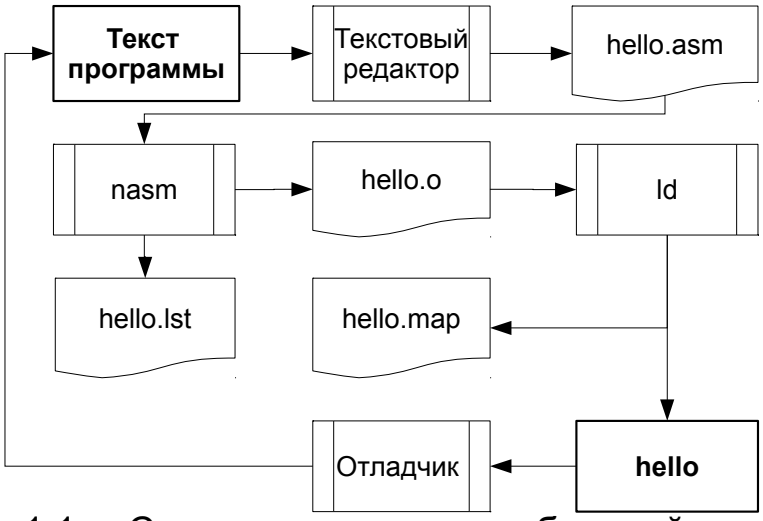


Рисунок 1.1 — Схема создания ассемблерной программы

1.1.4 Основные возможности текстового редактора mcedit

`mcedit` — это текстовый редактор, встроенный в двухпанельный файловый менеджер Midnight Commander. Сама по себе среда Midnight Commander (или просто `mc`) очень схожа с другими «командерами». Например, чтобы создать в текущем каталоге файл `lab1.asm` и начать его редактирование, можно набрать:

```
mcedit ./lab1.asm
```

Общий вид командной строки для запуска:

```
mcedit [-bcCdfhstVx?] [+число] file
```

Некоторые параметры:

+число	переход к указанной числом строке (не ставьте пробел между знаком + и числом)
-b	черно-белая цветовая гамма
-c	цветовой режим ANSI для терминалов без поддержки цвета
-d	отключить поддержку мыши
-V	вывести версию программы

`mcedit` — это полноценный полноэкранный редактор, позволяющий редактировать файлы размером до 64 Мб, с возможностью редактиро-

вания бинарных файлов. Основными возможностями являются: копирование блока, перемещение, удаление, вырезка, вставка; отмена; выпадающие меню; вставка файлов; макро-команды; поиск регулярных выражений и их замена; подсветка синтаксиса; перенос по словам; изменяемая длина табуляции; использование перенаправления потоков для применения, например, проверки орфографии при помощи ispell.

Редактор крайне прост в использовании и может быть использован без предварительного изучения. Выпадающее меню вызывается клавишей F9. Список наиболее часто используемых горячих клавиш приведен ниже (Ctrl и Shift обозначают соответствующие клавиши клавиатуры, Meta — условное обозначение для набора мета-клавиш, на современном компьютере это обычно Alt или Esc):

F3	Начать выделение текста. Повторное нажатие F3 закончит выделение
Shift+F3	Начать выделение блока текста. Повторное нажатие F3 закончит выделение
F5	Скопировать выделенный текст
F6	Переместить выделенный текст
F8	Удалить выделенный текст
Meta+l	Переход к строке по её номеру
Meta+q	Вставка литерала (непечатного символа). См. ниже
Meta+t	Сортировка строк выделенного текста
Meta+u	Выполнить внешнюю команду и вставить в позицию под курсором её вывод
Ctrl+f	Занести выделенный фрагмент во внутренний буфер обмена mc (записать во внешний файл)
Ctrl+k	Удалить часть строки до конца строки
Ctrl+n	Создать новый файл
Ctrl+s	Включить или выключить подсветку синтаксиса
Ctrl+t	Выбрать кодировку текста
Ctrl+u	Отменить действия
Ctrl+x	Перейти в конец следующего слова
Ctrl+y	Удалить строку
Ctrl+z	Перейти на начало предыдущего слова
Shift+F5	Вставка текста из внутреннего буфера обмена mc (прочитать внешний файл)
Meta+Enter	Диалог перехода к определению функции
Meta+-	Возврат после перехода к определению функции
Meta++	Переход вперед к определению функции
Meta+n	Включение/отключение отображения номеров строк
tab	Отодвигает вправо выделенный текст, если выключена опция «Постоянные блоки»

Meta-tab	Отодвигает влево выделенный текст, если выключена опция «Постоянные блоки»
Shift+Стрелки	Выделение текста
Meta+Стрелки	Выделение вертикального блока
Meta+Shift+-	Переключение режима отображения табуляций и пробелов
Meta+Shift++	Переключение режима «Автовыравнивание возвратом каретки»

Также работают и привычные по Norton и Volcov Commander'ам клавиши:

Ctrl-Ins	копировать
Shift-Ins	вставить
Shift-Del	вырезать
Ctrl-Del	удалить выделенный текст.

Выделение мышью также работает на некоторых терминалах.

Клавиши автозавершения (обычно Alt-Tab или Escape Tab) завершают слово, на котором находится курсор, используя ранее применявшиеся в файле слова.

Для задания макроса нажмите Ctrl-R и нажимайте клавиши, которые нужны для воспроизведения в будущем. Повторное нажатие Ctrl-R завершит запись макроса. Затем нажмите на клавишу, на которую хотите повесить этот макрос. Макрос сохранится, когда нажмете Ctrl-A и затем назначенную макросу клавишу. Макрос выполнится по нажатию Meta, Ctrl, или Esc назначенной клавиши, если клавиша не используется другими функциями.

Дополнительную информацию, как обычно в Linux, можно получить при помощи команды `man mc`.

1.1.5 Правила оформления ассемблерных программ

При наборе программ на языке ассемблера придерживайтесь следующих правил:

- директивы набирайте большими буквами, инструкции - малыми;
- пишите текст широко;
- не выходите за край экрана - его неудобно будет редактировать и печатать;
- для отступов пользуйтесь табуляцией (клавиша TAB);
- блоки комментариев задавайте с одинаковым отступом.

Оптимальной считается такая строка:

<TAB><TAB>mov<TAB>eax,<пробел>ebx<(1-3)TAB>
;<пробел>текст комментария

Количество табуляций перед комментарием определяется длиной аргументов команды и может быть от 1 до 3.

По мере знакомства с синтаксисом языка будут приводиться дополнительные правила.

1.1.6 Транслятор NASM

NASM превращает текст программы в объектный код. Имя программы задается в командной строке. В простейшем случае это выглядит так: `nasm hello.asm` (расширение указывать обязательно). Текст программы из файла `hello.asm` преобразуется в объектный код, который запишется в файл `hello.o`. Т. о. имена всех файлов получаются из имени входного файла и расширения по умолчанию.

NASM всегда создает выходные файлы в *текущем* каталоге.

NASM не запускают без параметров, т. к. он — всего лишь транслятор, а не интегрированная среда разработки.

Рекомендуется в рабочем каталоге все файлы хранить в определенной иерархии. Например, для первой работы создайте каталог `~/labs/asm/01`.

Для запуска транслятора достаточно набрать `nasm hello.asm`. При этом вы не увидите никаких сообщений — они появляются только в случае ошибок или предупреждений. При наличии ошибок объектный файл не создается.

Например, для компиляции приведенного выше текста программы «Hello World» необходимо писать:

```
nasm -f elf hello.asm
```

Ключ -f указывает транслятору создавать бинарные файлы в формате ELF (если используется 64-битная версия Linux, следует вместо elf указывать elf64 для генерации 64-битного кода).

Более подробно синтаксис командной строки рассмотрен в следующих работах.

1.1.7 Компоновщик LD

Как видно из схемы на рис. 1.1, чтобы получить исполняемую программу, объектный файл необходимо передать на обработку компоновщику (или, как его еще называют, линковщику):

```
ld -o hello hello.o
```

Ключ -o с последующим значением задает в данном случае имя создаваемого исполняемого файла.

Формат командной строки LD подробно рассмотрен в следующих работах, также его можно увидеть, набрав `ld --help`. Для получения более подробной информации см. `man ld`. Запустить на выполнение созданный исполняемый файл можно, набрав в командной строке:

```
./hello
```

Примечание: в данном случае исполняемый файл hello выполняется из текущего каталога (что обеспечивают символы «./» перед его именем).

1.2 Задание для выполнения

1. Создайте в своем домашнем каталоге новый подкаталог с именем `asm_01`. Создайте в нем с помощью редактора mcedit текстовый файл `lab1.asm`, и введите в него текст программы из п. 1.1.1, пользуясь правилами оформления ассемблерных программ.
2. Оттранслируйте полученный текст программы в объектный файл.
3. Выполните линковку объектного файла и запустите получившийся исполняемый файл.
4. Измените в тексте программы выводимую на экран строку с Hello world! на свою фамилию. Повторите пункты 2 и 3.

1.3 Содержание отчета

Отчет по лабораторной работе должен содержать:

- титульный лист;
- название, цель и краткое задание по лабораторной работе;
- схему алгоритма программы;
- текст программы на ассемблере;
- выводы по проделанной работе.

1.4 Контрольные вопросы

1. Как обрабатываются блоки текста в редакторе `mcedit`?
2. Как восстановить в `mcedit` удаленные строки?
3. Какие основные отличия ассемблерных программ от ЯВУ?
4. В чем отличие инструкции от директивы?
5. Каковы правила оформления программ на языке ассемблера?
6. Каковы этапы получения выполняемого файла?
7. Каково назначение этапа трансляции?
8. Каково назначение этапа компоновки?
9. Какие файлы могут создаваться при трансляции программы, какие из них создаются по-умолчанию?
10. Каковы форматы файлов для `nasm` и `ld`?

2 Дополнительные приемы трансляции программы на языке ассемблера

2.1 Краткие теоретические сведения

2.1.1 Системные инструменты

Расширенный синтаксис командной строки NASM и LD

Полный вариант командной строки nasm выглядит следующим образом:

`nasm` *[`-@` косвенный_файл_настроек] [`-o` объектный_файл][`-f` формат_объектного_файла] [`-l` листинг] [параметры...] [--] исходный_файл*

Для более подробной информации см. `man nasm`. Для получения списка форматов объектного файла см. `nasm -hf`.

Пример `nasm -f elf64 -g -l main.lst main.asm` скомпилирует исходный файл `main.asm` в `main.o`, при это формат выходного файла будет `elf64`, и в него будут включены символы для отладки (`g`), вдобавок будет создан файл листинга `main.lst`.

Примечание: формат `elf64` позволяет создавать исполняемый код, работающий под 64-битными версиями Linux. Для 32-битных версий ОС указываем в качестве формата просто `elf`.

Компоновщик `ld` имеет следующий формат командной строки:

`ld` *[параметры] объектные_файлы...*

Для более подробной информации см. `ld --help` или `man ld`.

Пример `ld -Map main.map -o main main.o` создаст исполняемый файл `main` из объектного файла `main.o`, при этом создавая карту памяти в файл `main.map`.

Компоновщик `ld` не предполагает по умолчанию расширений для файлов. Но принято использовать следующие расширения:

- `o` для объектных файлов;
- *без расширения* для исполняемых файлов;
- `map` для файлов схемы программы;
- `lib` для библиотек.

Если имя выполняемого файла не определено, оно будет образовано из имени первого объектного файла.

Утилита MAKE

`make` — утилита GNU для обработки групп программ. В том числе она может управлять трансляцией и компоновкой. Поведение `make` описывается файлом `Makefile`, который должен присутствовать в текущем каталоге. Это текстовый файл, который может содержать:

- комментарии;
- правила;
- макроопределения.

Пример:

```
build:
nasm -f elf64 -g -l main.lst main.asm
ld -Map main.map -o main main.o
clean:
rm -f main main.o main.lst main.map
# здесь могут быть дополнительные команды
```

В примере созданы два правила с именами («целями») `build` и `clean`. Первое из них выполняет трансляцию и сборку программы `main.asm`, а второе удаляет все созданные предыдущим правилом файлы. Таким образом, если набрать в командной строке `make build`, то будет создан исполняемый файл из `main.asm`. Если набрать `make clean`, то проект будет очищен: будут удалены все файлы, создаваемые `nasm` и `ld`. Для более подробной информации см. `man make`.

Примечание: утилита make необычно требовательна к содержимому make-файла. Ей необходимо, чтобы команды в правилах (в отличие от целей) начинались с отступа, и отступ обязательно должен создаваться символом табуляции, а не пробелами. Редактор `mcedit` по умолчанию заменяет табуляции пробелами. Чтобы он создавал настоящую табуляцию, нужно в меню (F9) выбрать подпункт «Общая» пункта «Настройка» и убрать галочку в пункте «Симулировать неполную табуляцию».

Основы работы с Midnight Commander

Midnight Commander (или просто `mc`) — это программа, которая позволяет просмотреть структуру каталогов и выполнить основные операции по управлению файловой системой. Другими словами, это файловый менеджер. Если вы имеете опыт работы с другими подобными файловыми менеджерами, например Norton Commander (`nc`) в DOS или FAR в Windows, то вы легко сможете работать и с `mc`, поскольку основные особенности поведения и даже основные комбинации «горячих клавиш» у них совпадают.

Для активации оболочки MC достаточно ввести в командной строке `mc` и нажать клавишу ввод.

Хотя для управления файловой системой и вообще для работы с файлами можно использовать такие команды операционной системы, как `pwd`, `ls`, `cd`, `mv`, `mkdir`, `rmdir`, `cp`, `rm`, `cat`, `more` и т. д.,

многие операции с файлами удобнее делать с помощью программы Midnight Commander, которая наглядно представляет все выполняемые действия.

Наиболее часто выполняемые в Midnight Commander операции привязаны к функциональным клавишам <F1> – <F10>. Приведем сводку в виде таблицы.

Функциональная клавиша	Выполняемое действие
<F1>	Вызывает контекстно-зависимую подсказку
<F2>	Вызывает меню, создаваемое пользователем
<F3>	Просмотр файла, на который указывает подсветка в активной панели
<F4>	Вызов встроенного редактора для файла, на который указывает подсветка в активной панели
<F5>	Копирование файла или группы отмеченных файлов из каталога, отображаемого в активной панели, в каталог, отображаемый на второй панели. При копировании одного файла можно поменять его имя. Можно также указать имя каталога, куда будет производиться копирование (если надо скопировать в каталог, отличный от каталога, отображаемого во второй панели)
<F6>	Перенос файла или группы отмеченных файлов из каталога, отображаемого в активной панели, в каталог, отображаемый на второй панели. Как и при копировании, можно поменять имя файла или целевого каталога.
<F7>	Создание подкаталога в каталоге, отображаемом в активной панели
<F8>	Удаление файла (подкаталога) или группы отмеченных файлов
<F9>	Вызов основного меню программы (отображаемого над панелями)
<F10>	Выход из программы

Вызвать главное меню можно клавишей F9.

Выпадающее подменю «Команды» главного меню позволяет выполнить еще ряд операций по управлению файловой системой, а так-

же выполнить некоторые команды, изменяющие вид панелей Midnight Commander и отображаемую в панели информацию.

При обращении к команде меню «Дерево каталогов» выводится окно, отображающее структуру каталогов файловой системы.

Команда «Поиск файла» (горячие клавиши <Meta>+<?> или <Esc>,<?>) выпадающего меню «Команда» позволяет вам найти на диске файл с заданным именем. После выбора этой команды меню вначале запрашивается имя искомого файла и имя каталога, с которого необходимо начинать поиск. Нажав экранную кнопку «Дерево», вы можете выбрать начальный каталог поиска из дерева каталогов. В поле «Содержание» (Contents) можно задать регулярное выражение по правилам команды `egrep`. В частности, это означает, что перед символами, имеющими специальное значение для `egrep`, необходимо вставить символ «\», например, если вам нужно найти строку «strcmp (», вы должны указать шаблон поиска в виде «`strcmp \(`». Для того, чтобы начать поиск, нажмите экранную кнопку «Дальше». Во время поиска его можно приостановить кнопкой «Остановить» и продолжить по кнопке «Продолжить».

Список найденных файлов можно просматривать, перемещаясь с помощью клавиш <Стрелка вверх> и <Стрелка вниз>. Кнопка «Перейти» используется для перехода в каталог, в котором находится подсвеченный файл. Кнопка «Повтор» служит для задания параметров нового поиска.

Кнопка «Выход» служит для выхода из режима поиска.

Нажатие на кнопку «Панелизация» приведет к тому, что результаты поиска будут отображены на текущую активную панель, так что вы можете производить с выбранными файлами еще какие-то действия (просматривать, копировать, перемещать, удалять и так далее). После вывода на панель можно нажать <Ctrl>+<R> для возврата к обычному списку файлов.

Команда «Переставить панели» (<Ctrl>+<U>) меняет местами содержимое правой и левой панелей.

По команде «Отключить панели» (<Ctrl>+<O>) показывается вывод последней из выполнявшихся команд shell.

По команде «Сравнить каталоги» (<Ctrl-X>, <D>) сравнивается содержимое каталогов, отображаемых на левой и правой панелях.

Команда меню «История команд» выводит окно со списком ранее выполнявшихся команд. Подсвеченную строку из истории можно скопировать в командную строку оболочки (перемещение подсветки — клавишами <Стрелка вверх> и <Стрелка вниз>, копирование — по клавише <Enter>).

Команда меню «Фоновые задания» позволяет вам управлять фоновыми заданиями, запущенными из Midnight Commander (такими зада-

ниями могут быть только операции копирования и перемещения файлов). Используя эту команду меню или горячие клавиши <Ctrl>+<X>, <J>, вы можете остановить, возобновить или снять любое из фоновых заданий.

После выбора команды меню «Файл расширений» вы получаете возможность редактировать файл `mc.ext`, в котором можете связать с определенным расширением файла (окончанием имени после последней точки) программу, которая будет запускаться для обработки (просмотра, редактирования или выполнения) файла с таким расширением. Запуск выбранной программы будет осуществляться после установки подсветки на имя файла и нажатия клавиши <Enter>.

Команда «Файл меню» используется для редактирования пользовательского меню (которое появляется после нажатия клавиши <F2>).

Настройка оболочки Midnight Commander

Одно из важных достоинств `mc` — конфигурируемость. Поскольку получение выполняемого файла из ассемблерной программы выполняется ручным запуском нескольких программ, на практике этот процесс как правило автоматизируют, например с использованием утилиты `make`. Но конечно же есть и другие способы. Несколько вариантов возможной автоматизации предоставляет `mc`.

Первый способ — это использование упоминавшегося выше файла расширений. За каждым *типом* файла можно закрепить действия, которые будут выполняться при выборе нужного файла и нажатии на клавишу <Enter>. Это действие задается в специальном файле расширений (bindings) `mc.ext` (F9→Команда→Файл расширений). Он имеет текстовый формат и может быть отредактирован любым текстовым редактором. Но предпочтение лучше отдавать встроенному редактору `mc`, т. к. в этом случае на экран появится подсказка по внутреннему формату файла. Наиболее просто войти в режим редактирования через меню `mc`, выбрав F9→Команда→Файл расширений.

Второй способ — использование меню. В отличие от предыдущего варианта, тут можно выбрать различные действия для одного и того же файла. Например, можно просто получить выполняемый модуль, а можно получить и запустить его. Настройка меню хранится в текстовом файле `.mc.menu` или `.mc/menu`. Как и файлов расширений, файлов меню может быть много, они бывают основными и локальными. Для их настройки можно выбрать: F9 → Команда → Файл меню → Main/Local. Внутренний формат очень похож на формат файла расширения и тоже ясен из подсказки.

2.1.2 Элементы программирования

Описание инструкций MOV и INT

Инструкция языка ассемблера mov предназначена для дублирования данных источника в приемнике. В общем виде записывается

```
mov dst, src
```

где `dst` — приемник, `src` — источник.

Инструкция языка ассемблера INT предназначена для вызова прерывания с указанным номером. В общем виде записывается

```
int  n
```

где `n` — номер прерывания, принадлежащий диапазону 0–255. При программировании в Linux с использованием вызовов ядра (sys_calls) `n = 80h` (принято задавать в шестнадцатеричной системе счисления).

Вызывая инструкцию `int 80h`, мы выполняем системный вызов какой-либо функции ядра Linux. При этом происходит передача управления ядру операционной системы. Чтобы узнать, какую именно системную функцию нужно выполнить, ядро извлекает номер системного вызова из регистра `eax`. Поэтому перед вызовом прерывания необходимо поместить в этот регистр нужный номер — например, выполнить `mov eax, 3` для системного вызова номер 3. Многим системным функциям требуется передавать какие-либо параметры. По принятым в ОС Linux правилам, эти параметры помещаются в порядке следования в остальные регистры процессора: `ebx`, `ecx`, `edx` и т. д. Если системная функция должна вернуть значение, она помещает его в регистр `eax`.

Системные вызовы для обеспечения диалога с пользователем

Простейший диалог с пользователем требует наличия двух функций: вывода текста на экран и ввода текста с клавиатуры. Простейший способ вывести строку на экран — использовать системный вызов `write`, который аналогичен функции `write` из языка Си и предназначен для записи данных в файл. Этот системный вызов имеет номер 4, поэтому перед вызовом инструкции `int` необходимо поместить значение 4 в регистр `eax`. Первым аргументом `write`, помещаемым в регистр `ebx`, задается дескриптор файла. Для вывода на экран в качестве дескриптора файла нужно указать 1 (что означает «стандартный вывод», т. е. вывод на дисплей).

Вторым аргументом задается адрес выводимой строки (помещаем его в регистр `ecx`, например инструкцией `mov ecx, hello`). Строка может иметь любую длину. Последним аргументом (т.е. в регистре `edx`) должна задаваться максимальная длина выводимой строки (посмотрите, как это делалось в программе из предыдущей работы).

Для ввода строки с клавиатуры можно использовать аналогичный системный вызов `read`. Этот системный вызов имеет номер 3. Подробная информация о нем, предоставляемая командой `man 2 read` показывает, что его аргументы — такие же, как у вызова `write`, только для «чтения» из клавиатуры мы используем файловый дескриптор 0 (стандартный ввод) а не (1 — стандартный вывод — как было при выводе на дисплей). Возвращаемое через регистр `eax` значение функции `read` — количество прочитанных с клавиатуры символов.

2.2 Задание для выполнения

1. Создайте новый текстовый файл `asdfg.asm` и сохраните его в своем домашнем каталоге.
2. Пользуясь информацией п. 2.1.2 написать программу, работающую по следующему алгоритму:
   - вывести приглашение, типа "Введите строку:";
   - ввести строку с клавиатуры;
   - вывести введенную строку на экран.
3. Создайте в своем домашнем каталоге новый подкаталог и скопируйте в него созданный файл с текстом программы.
4. Скопируйте файл `asdfg.asm` в `lab02-1.asm`.
5. Оттранслируйте полученный текст программы в объектный файл по схеме `lab02-1.asm` $\rightarrow$ `q.o` и `asdfg.asm` $\rightarrow$ `w.o`.
6. Повторить результат предыдущего пункта с использованием косвенного файла.
7. Создать для MAKE файл с явными правилами получения выполняемых файлов двух написанных программ. Проверить работу MAKE, внося изменения в комментарии программы.
8. Настроить файл расширений так, чтобы для `asm` выполнялась трансляция, а для `obj` — компоновка с созданием файла карты загрузки.
9. Создать локальное меню для `mc`, в котором для клавиши F5 задать запуск полного процесса получения выполняемого файла из программы, на которую установлен курсор.

2.3 Контрольные вопросы

1. Каково назначение и формат косвенных командных файлов для NASM?
2. Каково назначение и формат файлов подсказки для LD?.
3. Каково назначение утилиты MAKE?
4. Где задаются правила поведения MAKE?

3 Работа с файлом листинга

3.1 Краткие теоретические сведения

3.1.1 Системные инструменты

Права доступа к файлам

Как многопользовательская операционная система, ОС Linux содержит механизм разграничения доступа к данным, позволяющий как защитить данные одного пользователя от нежелательного вмешательства других, так и разрешить другим доступ к этим данным для совместной работы. Любой ресурс компьютера под управлением ОС Linux представляется как файл.

По отношению к файлу пользователь может входить в одну из трех категорий: *владелец*, *член группы владельца*, *все остальные*. Для каждой из этих категорий есть свой набор прав доступа. Первым владельцем файла становится его создатель. Дальше файл можно передать другому владельцу или в другую группу командой

```
chown [ключи] <новый_пользователь>[:новая_группа] <файл>
```

или

```
chgrp [ключи] < новая_группа > <файл>
```

Набор прав доступа задается тройками битов и состоит из прав на чтение, запись и исполнение файла. В символьном представлении он выглядит как строка «`rwx`», где вместо любого символа может стоять дефис. Буква означает наличие права (установлен в единицу второй бит триады `r` — чтение, первый бит `w` — запись, нулевой бит `x` — исполнение), а дефис означает отсутствие права (нулевое значение соответствующего бита). Очевидно, что эти три бита могут быть записаны еще и как восьмеричное число. Так, права доступа `r-x` (чтение и исполнение без записи) понимаются как три двоичные цифры 101 или как восьмеричная цифра 5. Численное представление прав доступа называется абсолютным, или двоичной маской.

Полная строка прав доступа в символьном представлении устроена так:

```
<права_владельца><права_группы><права_остальных>
```

В абсолютном представлении права владельца являются старшей цифрой восьмеричного числа, права группы — средней и права остальных — младшей. Так, права `rwxr-x--x` выглядят как двоичное число 111 101 001, или восьмеричное 751.

Команда изменения прав доступа `chmod` понимает как абсолютное, так и символьное указание прав.

Свойства (атрибуты) файлов и каталогов можно вывести на терминал с помощью команды `ls` с ключом `-l`:

```
$ls -I /home/deb/README
-rwxr-xr-- 1 deb users 0 Feb 14 19:08 /home/deb/README
```

Назначим файлу `/home/deb/README` права `rw-r`, то есть разрешим себе чтение и запись, группе только чтение, остальным пользователям — ничего:

```
$cd ~ # переход в свой домашний каталог
$chmod 640 README # 110 100 000 == 640
$ls -1 README
-rw-r 1 deb users 0 Feb 14 19:08 /home/deb/README
```

В символьном представлении можно явно указывать, кому какое право мы хотим добавить, отнять или присвоить. Добавим право на исполнение файла README группе и всем остальным:

```
$chmod go+x README
$ls -1 README
-rw-r-x--x 1 deb users 0 Feb 14 19:08 /home/deb/README
```

Формат символьного режима:

`chmod <категория><действие><набор_прав>< ƒайл >`

Возможные значения аргументов команды представлены в таблице:

Аргумент		**Значение**
Категория	u	Владелец
	g	Группа владельца
	o	Прочие
	a	Все пользователи, то есть «a» эквивалентно «ugo»
Действие	+	Добавить набор прав
	-	Отменить набор прав
	=	Назначить набор прав
Право	r	Право на чтение
	w	Право на запись
	x	Право на исполнение
	s	Право смены идентификатора пользователя или группы
	t	Бит прилипчивости (sticky-бит)
	u	Такие же права, как у владельца
	g	Такие же права, как у группы
	o	Такие же права, как у прочих

Название бита прилипчивости унаследовано от тех времен, когда объем оперативной памяти был маленьким, а процесс подкачки медленным. Этот бит позволял оставлять небольшие часто используемые программы в памяти для ускорения их запуска. Сейчас его значение переосмыслено: этот бит, установленный для каталога, приводит к тому, что удалять файлы из этого каталога могут только владелец файла

и владелец каталога. Обычно это используется в каталогах, открытых для записи всем (например, /tmp).

Права смены пользователя и группы (SUID-бит и SGID-бит) означают следующее. Обычно исполняемый файл (программа или командный сценарий) получает те же права на доступ к файлам, что и пользователь, который запустил его на выполнение. Но у этого файла есть еще и владелец, полномочия которого могут быть совсем другими. Наличие одного из этих битов позволяет выполняющейся программе пользоваться полномочиями владельца программного файла или члена его группы.

Назначение файла листинга

Листинг — это один из выходных файлов, создаваемых транслятором. Он имеет текстовый вид и нужен при отладке программы, т. к. кроме строк самой программы содержит дополнительную информацию.

Обычно nasm создает в результате ассемблирования только объектный файл. Получить файл листинга можно, указав ключ -l и задав имя файла листинга в командной строке. Например:

```
nasm -l main.lst main.asm
```

Рассмотрим фрагмент файла листинга.

```
1 BITS32 ; Говорим компилятору, что код 32-битный
2                   section .data
3 00000000 48656C6C6F20776F72- hello:     db 'Hello worl
d!',10
4 00000009 6C64210A
5                   helloLen:  equ $-hello ; Длина строк
и
6
7                       section .text
8                       global _start
9 ; чтобы линкер смог метку найти и сделать точкой вхо
да в программу.
10
11   start:
12 00000000 B804000000 mov eax,4 ; Системный вызов для з
аписи (sys_write)
13 00000005 BB01000000 mov ebx,1 ; Описатель файла 1 -
стандартный вывод
14 0000000A B9[00000000] mov ecx,hello ; Адрес строки he
llo в ecx
15 0000000F BA0D000000   mov edx,helloLen ; helloLen - э
то константа
```

```
16     ;  mov edx,[helloLen] для получения действительно
го значения
17 00000014 CD80        int 80h    ; Вызов ядра
18
19 00000016 B801000000 mov eax,1  ; Системный вызов для
 выхода (sys_exit)
20 0000001B BB00000000 mov ebx,0  ; Выход с кодом возвр
ата 0 (без ошибок)
21 00000020 CD80 int 80h    ; Вывозов ядра
```

Структура листинга

Строки в первой части листинга имеют следующую структуру (рис. 3.1):

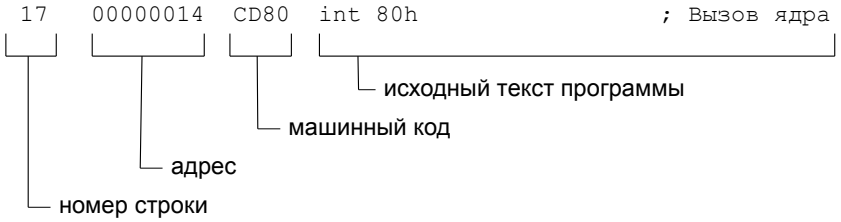


Рисунок 3.1 — Представление информации в строке листинга

Все ошибки и предупреждения, обнаруженные при ассемблировании, транслятор выводит на экран и файл листинга не создается.

- «Номер строки» представляет собой номер строки файла листинга. Номера строк особенно полезны при работе с перекрестными ссылками. Учтите, что номера строк в поле «номер строки» — это не номера строк исходного модуля. Например, при расширении макрокоманды или включении файла отсчет строк продолжается, хотя текущая строка в исходном файле остается той же. Чтобы перевести номер строки (сгенерированный, например, при создании перекрестных ссылок), вы должны найти соответствующую строку в листинге, а затем (по номеру или на глаз) найти ее в исходном файле.
- «Адрес» — это смещение машинного кода от начала текущего сегмента.
- «Машинный код» представляет собой действительную последовательность шестнадцатеричного значения байт и слов, которые ассемблируются из соответствующей исходной строки программы. Например, инструкция `int 80h` начинается по смещению 00000014 в сегменте кода. Информация справа от данной инструкции — это машинный код, в который ассемблируется инструкция, то есть инструкция `int 80h` ассемблируется в CD80 (в

шестнадцатеричном представлении). CD80 — это инструкция на машинном языке, вызывающая прерывание ядра.

- Наконец, поле «исходный текст программы» — это просто строка исходной программы вместе с комментариями. Некоторые строки на языке ассемблера (например, строки, содержащие только комментарии) не генерируют никакого машинного кода, и поля «смещение» и «исходный текст программы» в таких строках отсутствуют. Тем не менее номер строки им присваивается.

Допустимыми символами в метках являются буквы, цифры, _, $, #, @, ~, ., и ?. Начинаться метка или идентификатор могут с буквы, ., _ и ?. Перед идентификаторами, которые пишутся как зарезервированные слова нужно писать $, чтобы компилятора трактовал его верно. Максимальная длина идентификатора 4095 символов.

3.1.2 Элементы программирования

Описание инструкции вычитания

Схема команды целочисленного вычитания sub выглядит следующим образом:

```
sub операнд_1, операнд_2
```

Алгоритм работы команды включает два действия:

- выполнить вычитание: операнд_1 = операнд_2 – операнд_1
- установить флаги.

Флаги, устанавливаемые командой, подробнее рассматриваются ниже.

Команды условного перехода

Все машинные команды условного перехода, кроме одной, вычисляют условие перехода, анализируя один, два или три флага из регистра флагов, и лишь одна команда условного перехода вычисляет условие перехода, анализируя значение регистра CX. Команда условного перехода в языке Ассемблер имеет вид

```
j<мнемоника перехода> i8
```

Мнемоника перехода (от одной до трёх букв) связана со значением анализируемых флагов (или регистра CX), либо со способом формирования этих флагов. Чаще всего программисты формируют флаги, проверяя отношение между двумя операндами op1 <отношение> op2, для чего выполняется команда вычитания или команда сравнения. Команда сравнения имеет мнемонический код операции cmp и такой же формат, как и команда вычитания:

```
cmp op1,op2
```

Она и выполняется точно так же, как команда вычитания — за исключением того, что разность не записывается на место первого операнда. Таким образом, единственным результатом команды сравнения является формирование флагов, которые устанавливаются так же, как и при выполнении команды вычитания.

Программист по своему желанию может трактовать результат вычитания (сравнения) как производимый над знаковыми или же беззнаковыми числами. От этой трактовки может зависеть, будет ли один операнд считаться большим, чем другой, или же нет. Так, например, рассмотрим два коротких целых числа 0FFh и 01h — с учетом того, что отрицательные числа представляются процессором в дополнительном коде. Если числа знаковые, 0FFh = -1 < 01h = 1, а если беззнаковые, то 0FFh = 255 > 01h = 1.

Инструкции условной передачи управления бывают следующими:

Мнемокод	условие перехода	
	Флаги	Смысл
`ja/jnbe`	CF or ZF=0	выше /не ниже и не равно
`jae/jnb`	CF=0	выше или равно/не ниже
`jb/jnae`	CF=1	ниже/не выше и не равно
`jbe/jna`	CF or ZF=1	ниже или равно/не выше
`je/jz`	ZF=1	равно/нуль
`jne/jnz`	ZF=0	не равно/не нуль
`jg/jnle`	(SF xor OF) or ZF=0	больше/не меньше и не равно
`jge/jnl`	SF xor OF=0	больше или равно/не меньше
`jl/jnge`	(SF xor OF)=1	меньше/не больше и не равно
`jle/jng`	((SF xor OF) or ZF)=1	меньше или равно/не больше
`jp/jpe`	PF=1	есть паритет/паритет четный
`jnp/jpo`	PF=0	нет паритета/паритет нечетный
`jc`	CF=1	перенос
`jnc`	CF=0	нет переноса
`jo`	OF=1	переполнение
`jno`	OF=0	нет переполнения
`jns`	SF=0	знак +
`js`	SF=1	знак -

Мнемоники, идентичные по своему действию, написаны в таблице через дробь (например, `ja` и `jnbe`). Программист выбирает, какую из них применить, чтобы получить более простой для понимания текст программы.

Примечание: термины «выше» («a» от англ. «above») и «ниже» («b» от англ. «below»)применимы для сравнения беззнаковых величин (адресов), а термины «больше» («g» от англ. «greater») и «меньше» («l» от англ. «lower») используются при учете знака числа. Таким образом, мнемонику инструкции `ja/jnbe` можно расшифровать как «jump (пе-

реход) if *above* (если *выше*) / if *not below equal* (если не *меньше или равно*)».

3.2 Задание для выполнения

1. Написать программу, работающую по следующему алгоритму:
   - вывести на экран запрос о времени дня, например, «Полдень прошел?»;
   - принять с клавиатуры ответ (Y/N);
   - если было введено N выдать сообщение «Доброе утро», в противном слу- чае — «Добрый день».
2. Получить файл листинга и внимательно ознакомиться с его форматом и содержимым.
3. В любой инструкции с двумя операндами удалить один операнд и проассемблировать программу с получением файла листинга. Какие выходные файлы создаются в этом случае? Что добавляется в листинге?
4. Подробно объяснить содержимое трех строк файла листинга по выбору.
5. Получить имя владельца и числовой код прав доступа для файлов, сгенерированных в результате выполнения работы. Расшифровать права доступа.
6. Изменить права доступа к исполняемому файлу, запретив его выполнение. Попытаться выполнить файл.
7. Разрешить выполнение исходного текста программы как исполняемого файла. Попытаться выполнить его и объяснить результат.

3.3 Контрольные вопросы

1. Каким образом в Unix-подобных ОС определяются права доступа к файлу?
2. Как ОС определяет, является ли файл исполняемым? Как регулировать права на чтение и запись?
3. Как разграничить права доступа для различных категорий пользователей?
4. Для чего нужен файл листинга? В чем его отличие от текста программы?
5. Каков формат файла листинга? Из каких частей он состоит? Каково назначение первой части?
6. Как в программах на ассемблере можно выполнить ветвление?

4 Отладчик GDB

4.1 Краткие теоретические сведения

4.1.1 Понятие об отладке

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

- обнаружение ошибки;
- поиск ее местонахождения;
- определение причины ошибки;
- исправление ошибки.

Мы обнаруживаем ошибку когда программа «зависает» во время работы, или когда случается неустранимый сбой и операционная система принудительно завершает работу программы (т. н. «крах»), или если в процессе работы на экран выдается неверная (не такая, как ожидалось) информация. Некоторые ошибки обнаружить труднее: например, когда отличия от правильного результата не бросаются в глаза, или программа работает хорошо, пока не будет введено какое-то определенное значение (например, 0 или отрицательное число). Иногда только после тщательной проверки можно обнаружить, что результат отличается от ожидаемого в полтора раза, или, что в середине списка имен стоят неправильные инициалы.

Второй этап — поиск местонахождения ошибки — часто является самым трудным. Лучший способ найти место в программе, где находится ошибка — это разбивать программу на части и отлаживать их отдельно друг от друга. Структурное программирование идеально подходит для такой отладки.

Третий этап — выяснение причины ошибки — часто занимает второе место по трудности. После того, как будет определено местонахождение ошибки, обычно проще определить причину неправильной работы программы. Например, если вы определили, что ошибка находится в процедуре с именем PrintNames, вам достаточно просмотреть текст только этой процедуры, а не всей программы. Но даже в этом случае ошибка может оказаться настолько неуловимой, что вам придется поэкспериментировать, чтобы ее найти.

Последний этап — исправление ошибки. Вооружившись знанием языка программирования и знанием местонахождения ошибки, вы устраняете ее. После этого вы снова запускаете программу, обнаруживаете следующую ошибку, и все начинается снова. При написании программы последовательность этих этапов повторяется многократно.

4.1.2 Методы отладки

Наиболее часто применяют следующие методы отладки:

- создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые *диагностические сообщения*);
- использование специальных программ — отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить ее исправление. Наиболее популярные способы работы с отладчиком — это использование *точек останова* и *выполнение программы по шагам*.

Пошаговое выполнение — это выполнение программы с остановкой после каждой строчки, чтобы программист мог проверить значения переменных и выполнить другие действия.

Точки останова — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждет команд. Наиболее популярные виды точек останова:

- Breakpoint — собственно, точка останова. Остановка происходит, когда выполнение доходит до определенной строки, адреса или процедуры/функции.
- Watchpoint — точка просмотра. Выполнение программы приостанавливается, если программа обратилась к определенной переменной — либо считала ее значение, либо изменила его.

Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы — т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы.

4.1.3 Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки. В этой работе мы познакомимся с самыми основными возможностями GDB.

Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент ее выполнения, или что делает программа в момент краха.

GDB может выполнять действия четырех основных видов:

- начать выполнение программы, задав все, что может повлиять на ее поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась;
- изменить программу, так чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

Запуск отладчика GDB; выполнение программы; выход

Синтаксис команды для запуска отладчика имеет следующий вид:

```
gdb [опции] [имя_файла | ID процесса]
```

После запуска видим текстовое сообщение — т. н. «nice GDB logo». В следующей строке приглашение `(gdb)` ждет ввода команды.

Ниже приведен список некоторых команд GDB.

Краткую справку о любой команде можно получить, введя

```
(gdb) help [имя_команды, можно краткое]
```

Если при запуске GDB имя исполняемого файла, который нужно отладить, не было указано (что следовало бы делать), то указать его можно командой `file`:

```
(gdb) file <имя исполняемого файла для отладки>
```

Команда `run` (сокращенно `r` — многие команды GDB имеют краткую форму в виде одной или нескольких букв) запускает отлаживаемую программу под GDB. Если требуется, то после команды можно указать список аргументов программы. Также допускается перенаправление потоков ввода и вывода в другие файлы, например

```
(gdb) run > outfile
```

Если никаких точек останова не определено, то программа выполняется молча, при этом в начале и в конце выводятся сообщения:

```
(gdb) run
Starting program: test
Program exited normally.
(gdb)
```

Если же отладчик встречает точку останова, он выдает ее номер, адрес и дополнительную информацию — текущую строку, имя процедуры, и т.п.

Команда `kill` (`k`) прекращает отладку программы. Следует запрос

```
Kill the program being debugged? (y or n) y
```

Если в ответ введено `y` (то есть «да»), отладка программы прекращается. Командой `run` ее можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.

Выход из отладчика — команда `quit` (`q`):

```
(gdb) q
```

Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, программу можно отлаживать, работая в отладчике непосредственно с ее исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

Однако такая возможность есть не всегда, и в случае необходимости отладчик может дизассемблировать исполняемый код, изображая машинные команды в виде ассемблерных мнемоник.

Существуют два режима отображения синтаксиса машинных команд: режим Intel, используемый в т.ч. в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду

```
(gdb) set disassembly-flavor intel
```

Точки останова

Примечание: Информацию о командах этого раздела можно получить, введя

```
help breakpoints
```

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звездочка»:

```
(gdb) break *0x4000b5
```

В данном случае была установлена точка останова по адресу 4000b5, заданному в шестнадцатиричной системе счисления.

Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`). Команда `info` имеет много возможностей, но в данном случае воспользуемся лишь следующим ее форматом:

```
(gdb) info breakpoints
```

или, кратко

```
(gdb) i b
```

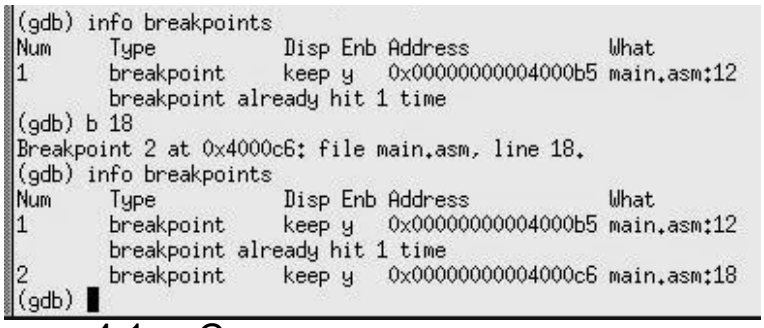


Рисунок 4.1 — Сессия отладки с точками останова

Если какая-то точка останова не нужна, то ее можно сделать неактивной с помощью команды `disable`:

```
disable breakpoint <номер этой точки>
```

Обратно, деактивированная точка останова активируется командой `enable`:

```
enable breakpoint <номер этой точки>
```

Статус точки останова (активна она или нет) легко узнать с помощью той же команды `info`. Если же точка останова не требуется вообще, она может быть удалена:

```
(gdb) delete breakpoint [номер точки]
```

а короче

```
(gdb) d b [номер точки]
```

Ввод этой команды без аргумента удалит *все* точки останова.

Возобновление выполнения, пошаговая отладка

Информацию о командах этого раздела можно получить, введя

```
(gdb) help running
```

Команда `continue` (`c`) продолжает выполнение остановленной программы:

```
(gdb) c [аргумент]
```

Выполнение будет происходить, пока снова не встретится точка останова. В качестве аргумента может использоваться целое число N. Это укажет отладчику проигнорировать (N - 1) точку останова (выполнение остановится на N-ой).

Команда `step` (`s`) приводит к выполнению программы до тех пор, пока не будет достигнута следующая строка ее кода:

```
(gdb) s [аргумент]
```

При указании аргумента — целого числа N, отладчик выполняет команду `step` N раз (если не останавливает выполнение из-за точек останова или по иным причинам).

Команда `next` (`n`) действует почти как `step`, но вызов процедуры считается единой инструкцией:

```
(gdb) n [аргумент]
```

Аргумент N работает так же, как и для `step`.

Команда `finish` (`fin`) выполняет программу до момента выхода из текущей процедуры (функции):

```
(gdb) fin
```

Если функция возвращает значение, то это значение выводится тоже.

Команда `until` (`u`) производит выполнение программы до тех пор, пока не будет достигнута строка с номером, большим текущего. Команду `until` удобно применять при отладке циклов. Остановка произойдет также, если программа при выполнении цикла выйдет из текущей процедуры, функции.

Команда `stepi` (`si`) по своему действию подобна `step`, но выполняется не строка, а ровно одна машинная инструкция:

```
(gdb) si [аргумент]
```

Аргумент N нужен, если требуется выполнить N инструкций.

Команда `nexti` (`ni`) аналогична `stepi`, но вызов процедуры трактуется отладчиком как одна инструкция, а не как передача управления на еще один блок ассемблерного кода, который тоже должен быть пройден по шагам:

```
(gdb) ni [аргумент]
```

Примечание: при использовании дизассемблированной программы, скомпилированной без информации для отладки, нет возможности выполнить «одну строку исходного кода» за отсутствием такового. В этом случае используются команды `stepi` и `nexti`.

Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти, а при необходимости позволяет вручную изменять значения регистров и переменных.

Для исследования содержимого памяти можно использовать команду `x`.

`x адрес` выдает содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/nfu адрес`. Рассмотрим задание формата подробнее.

- `n` — счетчик повторений. Это десятичное целое число, по умолчанию 1. Он определяет, сколько ячеек памяти отобразить (считая в единицах `u`).
- `f` — формат отображения. Наиболее употребляемые варианты: `s` (строка, оканчивающаяся нулем), `i` (машинная инструкция), `x`

(шестнадцатеричное число (значение по умолчанию). Значение по умолчанию изменяется каждый раз, когда вы используете команду с указанным в ней форматом.

- `u` — размер отображаемых ячеек памяти. Наиболее распространены варианты `b` (байт), `h` (полуслово, 2 байта), `w` (машинное слово, 4 байта, принято по умолчанию), `g` (длинное слово, восемь байт).

Например, `x/3uh 0x54320` — запрос на вывод трех полуслов (`h`) памяти в формате беззнаковых десятичных целых (`u`), начиная с адреса 0x54320.

С помощью команды `x` можно отображать также значения регистров (перед именем регистра обязательно ставится префикс `$`), например, `p/x $ax`.

Если необходимо часто выводить значение какого-либо выражения (чтобы увидеть, как оно меняется), вы можете добавить его в *список автоматического отображения*, чтобы GDB выводил его значение каждый раз при остановке вашей программы. Для этого служит команда `display`, которой можно передавать такой же аргумент, как команде `x`. Чтобы убрать элемент данных с заданным номером из списка отображения, можно использовать команду `delete display номера`.

И наконец, изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си):

```
set  {char}0x600155='s'
```

4.2 Элементы программирования

4.2.1 Индексный доступ к данным

Для индексного доступа применяют косвенную адресацию. В этом случае адрес ячейки памяти заносят в регистр и используют его в качестве указателя. Для этих целей можно применять любые регистры. Признаком косвенной адресации являются прямоугольные скобки. Пример использования косвенной адресации:

```
;
                              ; — регистровая адресация —
    mov     eax, esi          ; значение регистра esi
                              ; дублируется в eax

                              ; — косвенная адресация —
    mov     eax, [esi]        ; слово, расположенное по адресу esi,
                              ; дублируется в регистре eax
```

4.2.2 Инструкции организации циклов

Для организации циклов существуют специальные инструкции. Для всех них максимальное количество проходов задается в регистре `ecx`. Наиболее простой является инструкция `loop`. Она позволяет организовать безусловный цикл:

```
;
        mov     ecx, 100        ; количество проходов
        mov     esi, Buf1       ; смещение на первый буфер
        mov     ebx, Buf2       ; на второй
NextStep:
        mov     al, BYTE [esi]  ; перенос байта из одного
        mov     BYTE [ebx], al  ; буфера в другой
        inc     esi             ; перемещение указателей к
        inc     ebx             ; следующим элементам буфера
        loop    NextStep        ; повторить ecx раз от метки NextStep
```

4.3 Задание для выполнения

1. Написать программу со следующим алгоритмом:
   - ввести с клавиатуры символьную строку в буфер;
   - изменить порядок следования символов в строке на противоположный; положение символа 10 (`\n`) остается без изменений;
   - вывести результат на экран;
   - завершить программу.
2. Загрузить программу в отладчик. Какими способами это можно сделать?
3. Просмотреть дизассемблированный код программы, введя команду
   `disassemble _start`
4. Переключить дизассемблер GDB с синтаксиса ATT на синтаксис Intel и снова выполнить показ дизассемблированного кода. Найти три отличия. Переписать в тетрадь адрес второй инструкции в формате 0x12345678.
5. Установить точку останова на второй инструкции, указав её адрес (записан в тетради).
6. Выполнить программу. Что произошло?
7. Выполнить программу по шагам.
8. Посмотреть содержимое регистров в окне и с помощью команды `info r`.
9. Выполнить программу до места заполнения входного буфера. Вывести содержимое входного буфера в в шестнадцатеричном формате и в символьном виде (команда `x`).
10. Выполнить 2 прохода цикла по шагам, контролируя значения регистров. Какие регистры изменяются в цикле?
11. Изменить число проходов цикла на 5.

12. Изменить содержимое выходного буфера. Вводить данные как символы и как десятичные числа.

4.4 Контрольные вопросы

1. Какими возможностями обладает отладчик GDB?
2. Что такое дизассемблирование программы?
3. Что такое косвенная адресация?
4. Как организовать безусловный цикл?

5 Визуальная отладка; подпрограммы

5.1 Краткие теоретические сведения

5.1.1 Основные возможности отладчика EDB

Отладчик EDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент ее выполнения, или что делает программа в момент краха, однако реализует эти возможности через наглядный графический интерфейс. Синтаксис команды для запуска отладчика имеет следующий вид:

```
edb [ --attach <ID процесса>] [--run <имя_файла>
  (аргументы)]
```

После запуска появляется графическое окно EDB, разделенное на четыре основные части: дизассемблер, стек, дамп памяти с вкладками, содержимое регистров.

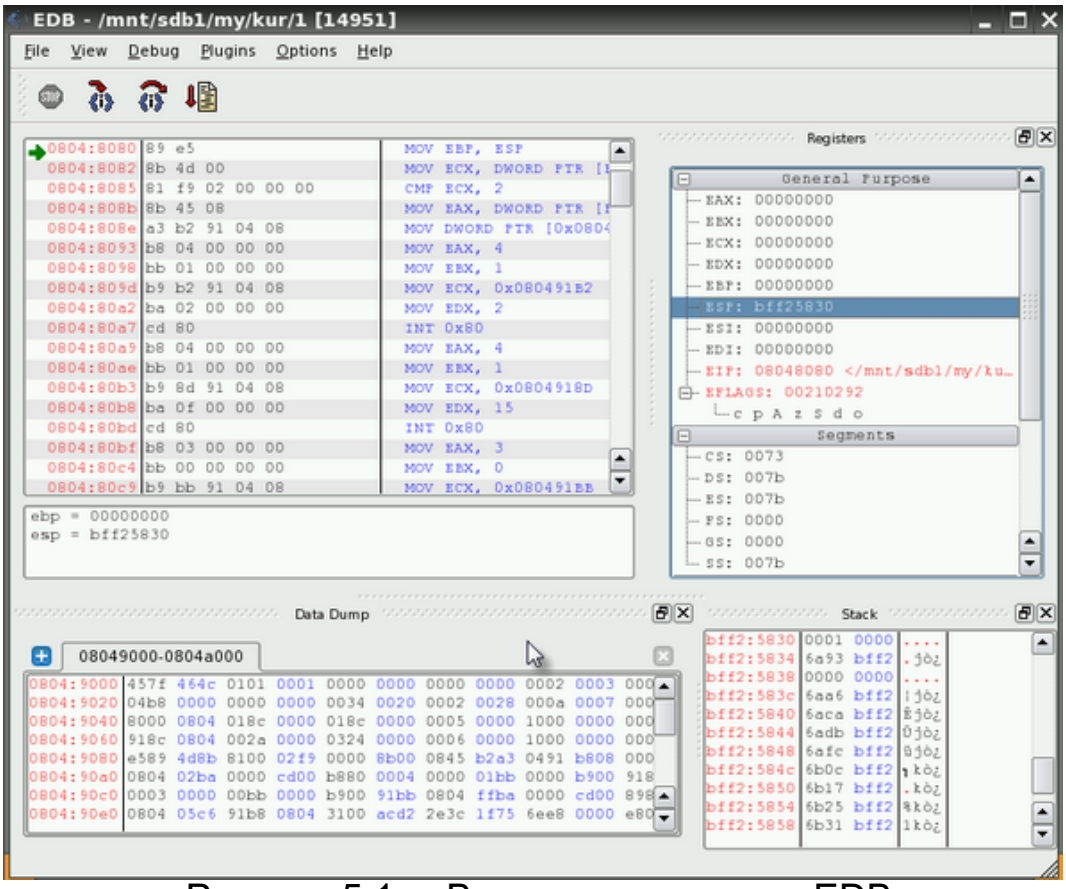


Рисунок 5.1 — Вид окна отладчика EDB

Всю информацию и данные Evan's Debugger отображает в меню и в окнах. Используются различные виды окон в зависимости от того, какого типа информация в них отображается. Все окна открываются и закрываются с помощью команд меню (или активных клавиш, соответствующих этим командам).

Окно регистров (Registers) отображает состояние регистров и флагов процессора, а также позволяет изменять их значения с помощью двойного нажатия мышкой. С помощью команд всплывающего (локального) меню можно попробовать перейти по адресу, хранящемуся в выбранном регистре, в окне стека или окне дампа памяти.

Окно дампа памяти (Data Dump) отображает построчное содержимое области памяти. Можно просматривать данные в виде шестнадцатеричных байтов, слов и двойных слов. Его можно использовать в тех случаях, когда желательно просмотреть некоторые исходные данные, не заботясь об остальном состоянии процессора. Во всплывающем меню имеются команды, которые позволяют модифицировать отображаемые данные, менять формат их отображения на экране и манипулировать блоками данных.

Окно стека (Stack) отображает текущее состояние стека, причем область первой вызванной функции будет находиться на дне стека, а всех последующих вызванных функций — в направлении вершины стека в последовательности их вызова. Во всплывающем меню имеются команды, которые позволяют модифицировать отображаемые данные, менять формат их отображения на экране, переходить по указанному адресу, по адресу из регистров `ebp` или `esp`.

Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, программу можно отлаживать, работая в отладчике непосредственно с ее исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

Однако такая возможность есть не всегда, и в случае необходимости отладчик может дизассемблировать исполняемый код, изображая машинные команды в виде ассемблерных мнемоник.

Напомним, что существуют два режима отображения синтаксиса машинных команд: режим Intel, используемый в т.ч. в NASM, и режим ATT. По умолчанию в дизассемблере EDB принят режим Intel.

Точки останова

Установить точку останова можно путем двойного клика мышкой на нужной инструкции. Если точка останова установилась, напротив инструкции появится красная отметка:

Информацию о всех установленных точках останова можно посмотреть с помощью Breakpoint Manager, нажав Ctrl+M или открыв меню Plugins $\rightarrow$ BreakpointManager $\rightarrow$ Breakpoints:

Рисунок 5.2 — Отображение точки останова

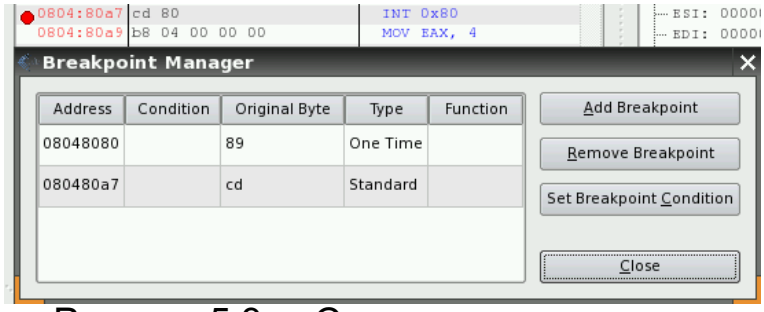


Рисунок 5.3 — Список точек останова

Возобновление выполнения, пошаговая отладка

Команда Run (F9) продолжает выполнение остановленной программы. Выполнение будет происходить, пока не встретится точка останова или программа не выполнится полностью.

Команда Step Into (F7) приводит к выполнению программы до тех пор, пока не будет достигнута следующая строка ее кода. Вызов процедуры трактуется отладчиком не как одна инструкция, а как передача управления на еще один блок ассемблерного кода, который тоже должен быть пройден по шагам.

Команда Step Over (F8) приводит к выполнению программы до тех пор, пока не будет достигнута следующая строка ее кода. В отличие от Step Into, вызов процедуры считается единой инструкцией.

5.2 Элементы программирования

5.2.1 Понятие подпрограммы

Подпрограмма — это, как правило, функционально-законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов, из подпрограмм существует возврат на команду, следующую за вызовом.

Если в программе встречаются одинаковый участки кода, их можно оформить в виде подпрограммы, а во всех нужных местах поставить ее вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер всей программы.

5.2.2 Инструкция call

Основные моменты выполнения подпрограммы иллюстрируются на рис. 5.4. В вызывающей подпрограмму программе выполняется ин-

струкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `eip` адрес соответствующей подпрограммы, осуществляя таким образом переход. После этого подпрограмма выполняется, как любой другой код. В подпрограммах могут (часто это так и бывает) содержаться инструкции вызовов других подпрограмм.

Рисунок 5.4 — Выполнение подпрограммы

Когда подпрограмма заканчивает работу, она вызывает инструкцию `ret`, которая извлекает из стека адрес, занесенный туда соответствующей инструкцией `call`, и заносит его в `eip`. Это приводит к тому, что вызывающая программа возобновит выполнение с инструкции, следующей за инструкцией `call`.

Например, следующая программа выводит на экран строку «Enter string:», ждёт ввода строки (например, «HELLO») и выводит на экран строку «Result:» и введённую строку (т. е. «Result: HELLO»). Для вывода строк вызывается подпрограмма `PrintString`:

```
    ;
SECTION .data               ; Константы
ask1:           DB    'Enter string: ', 10
ask1_len:       EQU   $-ask1
result:         DB    'Result: '
result_len:     EQU   $-result

SECTION   .bss
buf1:           RESB 80

SECTION .text               ; Код программы
     GLOBAL _start          ; Начало программы
;------------------------------------
; Подпрограмма вывода на экран строки
; Входные данные:
; ecx - указатель на выводимую строку.
; Нарушаемые регистры: eax,ebx;
;------------------------------------
 Print_string:
     mov eax, 4
     mov ebx, 1
     int 80h
     ret                    ; возврат в вызывающую программу

;------------------------------------
```

```
; Подпрограмма ввода строки с клавиатуры
; Входные данные:
; ecx - указатель на буфер для входной строки.
; Нарушаемые регистры: eax,ebx;
;-------------------------------
Enter_string:
     mov  eax, 3
     mov  ebx, 0
     int  80h
     ret                    ; возврат в вызывающую программу

_start:
     mov  ecx, ask1
     mov  edx, ask1_len
     call Print_string

     mov  ecx, buf1
     mov  edx, 6
     call Enter_string

     mov  ecx, result
     mov  edx, result_len
     call Print_string

     mov  ecx, buf1
     mov  edx, 6
     call Print_string

     mov  eax,1             ; Системный вызов для выхода (sys_exit)
     mov  ebx,0             ; Выход с кодом возврата 0 (без ошибок)
     int  80h               ; Вывзов ядра
```

Подпрограмма `PrintString` не настроена жестко на печать определенной строки. Она может печатать любую строку, на которую укажет регистр `ecx`.

5.2.3 Инструкция ret

Инструкция `ret` возвращает управление вызывающей программе. Для этого она извлекает из вершины стека четыре байта и заносит их в регистр счётчик команд `eip`. После этого значение регистра `esp` увеличится на 4. Если в процедуре занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата, и это приведёт к ошибке выхода из процедуры.

Ассемблерная подпрограмма без команды возврата не вернется в точку вызова, а будет выполнять следующий за подпрограммой код, как будто он является ее продолжением.

5.2.4 Способ перевода числа в десятичную символьную запись

Ввод информации с клавиатуры и вывод ее на экран осуществляется в символьном виде. Кодирование этой информации производится согласно кодовой таблице символов, где каждый символ (в простейшем случае) кодируется одним байтом. Однако в памяти компьютера любые числа, над которыми можно производить математические операции, записаны в двоичной системе счисления, и для вывода на экран необходимо преобразовать двоичное число в его символьную запись (а при вводе с клавиатуры — выполнить обратное преобразование).

Любое число X в позиционной системе счисления представляется в виде суммы произведений: $X = a_n \cdot p^n + a_{n-1} \cdot p^{n-1} + \ldots + a_0 \cdot p^0$. Здесь X — это число в системе с основанием p, имеющее $n+1$ цифру в целой части.

Так, при переводе кода введённого символа в десятичную систему (например, чтобы вывести на экран не символ, а численное значение его кода) надо разложить число на слагаемые, содержащие степени числа 10. Перевод кода символа производится путем последовательного деления на 10 с выделением остатков от деления до тех пор, пока частное не станет меньше делителя. Выписывая остатки от деления справа налево, получаем 10-чную запись числа.

5.2.5 Команды деления

Сначала рассмотрим деление 16-битового значения на 8-битовое. При беззнаковом делении 16-битового значения на 8-битовое, делимое должно быть записано в регистре `ax`. 8-битовый делитель может храниться в любом 8-битовом общем регистре или переменной в памяти соответствующего размера. Инструкция `div` всегда записывает 8-битовое частное в регистр `al`, а 8-битовый остаток — в `ah`. Например, после выполнения инструкций

```
;
        . . .
        mov    ax,51
        mov    dl,10
        div    dl
        . . .
```

результат 5 (51/10) будет записан в регистр `al`, а остаток 1 (остаток от деления 51/10) — в регистр `ah`.

Заметим, что частное представляет собой 8-битовое значение. Это означает, что результат деления 16-битового операнда на 8-битовый операнд не должен превышать 255. Если частное слишком велико, то генерируется прерывание 0 («деление на 0»). Инструкции

```
;
        . . .
        mov    ax,0fffh
        mov    bl,1
        div    bl
        . . .
```

генерируют прерывание по делению на 0 (как можно ожидать, прерывание по делению на 0 генерируется также, если 0 используется в качестве делителя).

При делении 32-битового операнда на 16-битовый операнд делимое должно записываться в регистрах `dx:ax`. 16-битовый делитель может находиться в любом из 16-битовых регистров общего назначения или в переменной в памяти соответствующего размера. Например, в результате выполнения инструкций

```
;
        . . .
        mov    ax,2
        mov    dx,1          ; загрузить в регистровую
        mov    bx,10h        ; пару dx:ax 10002h
        div    bx
    . . .
```

частное 1000h (результат деления 10002h на 10h) будет записано в регистре `ax`, а 2 (остаток от деления) — в регистре `dx`.

При делении имеет значение, используются операнды со знаком или без знака. Для деления беззнаковых операндов используется операция `div`, а для деления операндов со знаком — `idiv`.

5.3 Задание для выполнения

1. Написать программу со следующим алгоритмом:
   - ввести символ с клавиатуры;
   - преобразовать полученный код в десятичную символьную запись;
   - вывести символ и его код.

   Перевод числа в десятичную символьную запись оформить в виде подпрограммы.
2. Загрузить программу в отладчик. Это можно сделать двумя способами: написать в командной строке `edb --run имя_программы` или запустить `edb` и выбрать программу через пункт Open меню File.
3. Выполнить программу по шагам, нажимая кнопку Step Over панели инструментов или клавишу F7 (находясь в основном окне отладчика), до конца.
4. Поместить в программу точку останова на инструкции, следующей после ввода символа с клавиатуры — щелкнув правой кнопкой по нужной строке дизассемблированного кода и выбрав пункт

Add Breakpoint всплывающего меню. Выполнить программу до точки останова, нажав клавишу F9 или кнопку Run панели инструментов. Иметь в виду, что ввод текста с клавиатуры в выполняемую программу *осуществляется в отдельном окне* EDB Output, а не в основном окне отладчика.

5. Вывести в окне дампа памяти содержимое входного буфера, щелкнув в подокне Data Dump правой кнопкой мыши и выбрав пункт Goto Address всплывающего меню. Адрес вводить в шестрадцатиричной нотации Си (начиная с символов 0x).
6. Зайти в процедуру перевода числа в десятичную запись. Выполнить 2 прохода цикла по F7 (Step Into), контролируя значения регистров. Какие регистры изменяются в цикле?
7. Остальные проходы цикла выполнить по F8 (Step Over). В чем разница?
8. Определить физический адрес выходного буфера в ОЗУ .
9. Вывести ячейки памяти, соответствующие выходному буферу, в подокне Data Dump в шестнадцатиричном и в символьном виде.
10. Установить точку останова на инструкцию `div`. Выполнить программу, несколько раз нажав на F8 и наблюдая за изменением содержимого выходного буфера в подокне Data Dump. Каков результат? (Перевод чисел между шестнадцатиричной и десятичной системами счисления можно упростить, воспользовавшись программой Калькулятор, выбрав в ней пункт меню Вид $\rightarrow$ Программирование).
11. Изменить содержимое входного буфера и проверить, как это отражается на выполнении программы.

5.4 Контрольные вопросы

1. Как Evan's Debugger отображает информацию и данные?
2. Можно ли отлаживать исполняемый файл, если нет файла с исходным текстом программы?
3. Объясните, каким образом процессор осуществляет переход на подпрограмму, когда встречает инструкцию call?
4. Какие действия выполняет процессор, встретив инструкцию ret?
5. В каких случаях при делении генерируется прерывание 0?

6 Битовые операции

6.1 Краткие теоретические сведения

6.1.1 Команды сдвига

Команды сдвига позволяют выполнять действия над отдельными битами операндов. Все команды сдвига перемещают биты в поле операндов влево или вправо. При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита. Рассмотрим следующие команды сдвига:

`shr операнд, счетчик_сдвигов ;` логический (беззнаковый) сдвиг вправо

`shl операнд, счетчик_сдвигов ;` логический (беззнаковый) сдвиг влево

В следующем примере команда shr сдвигает содержимое регистра al вправо на 1 бит. Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра al заполняется нулем.

```
;
   mov   al,10110111b    ;в al содержится 10110111
   shr   al,1            ;в al содержится 01011011, CF=1
```

При сдвигах влево правые биты заполняются нулями.

Сдвиг влево на один двоичный разряд часто используется для удваивания чисел, а сдвиг на один разряд вправо — для деления на 2. Эти операции осуществляются значительно быстрее, чем команды умножения или деления.

Количество позиций, на которые должен быть выполнен сдвиг, задаются вторым аргументом инструкции сдвига, и может быть либо задано константой, либо регистром cl: например, `shl ax, 12` или `shl eax, cl`.

6.1.2 Команды циклического сдвига

Циклический сдвиг представляет собой операцию сдвига, при которой бит, выдвинутый с одного «конца» числа, занимает освободившийся разряд с другой стороны этого же числа. Существуют следующие команды циклического сдвига:

```
ror  ; Циклический сдвиг вправо
rol  ; Циклический сдвиг влево
rcr  ; Циклический сдвиг вправо с переносом
rcl  ; Циклический сдвиг влево с переносом
```

В командах rcr и rcl в сдвиге участвует флаг CF. Выдвигаемый из регистра бит заносится в флаг CF, а прежнее значение CF при этом поступает в освободившуюся позицию в регистре.

6.1.3 Команды работы с битами операндов

Команды `and`, `or`, `xor` и `test` являются командами логических операций. Эти команды имеют два операнда и используются для сброса, установки и проверки бит. Размерность операндов должна быть одинакова. Например, если размерность операндов равна слову (16 бит), то логическая операция выполняется сначала над нулевыми битами операндов, и далее над всеми битами с первого по пятнадцатый. Результат записывается на место первого операнда. Исключение составляет команда `test`. Команда `test` действует аналогично команде `and`, но результат не записывает на место первого операнда, а устанавливает только флаги во флаговом регистре. Это дает возможность анализировать отдельные биты не изменяя операнд.

Таблица истинности для	`and`	`or`	`xor`
Значение операнда 1	0101	0101	0101
Значение операнда 2	0011	0011	0011
Результат операции	0001	0111	0110

Команда `or`: Если хотя бы один из сравниваемых битов равен 1, то результат равен 1; если сравниваемые биты равны 0, то результат — 0. Команда `and`: Если оба из сравниваемых битов равны 1, то результат равен 1; во всех остальных случаях результат — 0. Команда `xor`: Если один из сравниваемых битов равен 0,а другой 1, то результат равен 1; если сравниваемые биты одинаковы (оба — 0 или оба — 1) то результат — 0.

Будем рассматривать байтную переменную `flags` как восемь независимых флагов, которые необходимо сбрасывать, устанавливать и анализировать в какой-либо программе. Для этого в правый операнд заносим «маску» в двоичной системе счисления. Рассмотрим пример установки второго бита с использованием команды `or`:

```
    ;
or   flags , 00000100b      ; второй бит = 1, остальные
                            ; без изменений
```

Сбросить второй бит можно командой `and`, но маска должна быть инверсная:

```
    ;
and  flags , 11111011b      ; второй бит = 0, остальные
                            ; без изменений
```

Для того чтобы проверить установлен ли нужный бит, применяется команда `test`:

```
    ;
test  flags , 100b          ; переменная flags не изменилась
jz    ...                   ; изменены ZF, SF, PF
```

Для сброса всех бит можно использовать команду `xor`:

```
    ;
xor  ax, ax                 ; ax — обнулен
```

Еще одна логическая команда `not` устанавливает обратное значение бит в байте или в слове, в регистре или в памяти: нули становятся единицами, а единицы — нулями. Если, например, регистр `al` содержит 1100 0101, то команда `not al` изменяет это значение на 0011 1010.

6.2 Задание для выполнения

Написать программу со следующим алгоритмом:

- вывести приглашение;
- ввести с клавиатуры строку (предполагается, что она содержит десятичные цифры и любые буквы);
- найти во введенной строке все цифры и для каждой найденной цифры установить в «1» в регистре `ax` бит, номер которого равен этой цифре;
- вывести на экран содержимое регистра `ax` в виде нулей и единиц;
- объяснить полученный результат.

6.3 Контрольные вопросы

1. Как работают инструкции сдвигов? В чем отличие циклических сдвигов?
2. Какие инструкции сдвигов часто используются для удваивания чисел, и для деления на 2?
3. Как проверить установлен ли нужный бит в переменной?
4. Какой инструкцией можно сбросить второй бит в переменной?

7 Ввод и вывод числовых данных

7.1 Краткие теоретические сведения

7.1.1 Ввод целых чисел с клавиатуры

При работе с числами, введенными с клавиатуры, необходимо учитывать следующий факт. Любые символы, вводимые пользователем (в т. ч. и цифры) представлены соответствующими ASCII-кодами. Например, когда пользователь вводит с клавиатуры цифру `'0'` — на самом деле считывается байт, равный ее ASCII-коду, т. е. 30h. Соответственно, цифра `'1'` будет представлена кодом 31h, цифра `'2'` — 32h и так далее.

При вводе одноразрядного десятичного числа ASCII-код соответствующей ему цифры можно превратить в нужное число, просто вычтя из него ASCII-код цифры ноль, т. е. 30h. Получить число из ASCII-записи, состоящей из нескольких цифр — более сложная задача. Рассмотрим один из возможных алгоритмов.

Как мы знаем, любое число X в десятичной системе счисления представляется в виде суммы произведений: $X = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \ldots + a_0 \cdot 10^0$ Здесь a_n — это число, соответствующее разряду.

Присваиваем ноль переменной , в которой должен быть сохранен результат:

$X = 0$.

Далее, в цикле последовательно обрабатываем все символы числа, начиная с самой старшей (т. е. проходим символьную запись десятичного числа слева направо).

$a_n =$ *ASCII-код*$\_a_n - 30h$ (Из ASCII-кода текущей цифры вычитаем 30h);

$X = X \cdot 10 + a_n$ (Умножаем содержимое переменной результата на 10, прибавляем к нему значение текущего разряда. Сохраняем получившееся значение обратно в переменную X.)

Таким образом, пройдя цикл для всех цифр числа, мы получим в переменной X числовое представление введенной с клавиатуры строки цифр.

7.2 Элементы программирования на языке ассемблера

7.2.1 Команды умножения

Инструкция `mul` перемножает 8- или 16- или 32-битные беззнаковые сомножители, создавая 16- или 32- или 64-битное произведение. Рассмотрим умножение 8-битных сомножителей.

При 8-битном (8-разрядном) умножении один из операндов должен храниться в регистре al, а другой может представлять собой любой 8-битовый общий регистр или переменную памяти соответствующего размера. Инструкция mul всегда сохраняет 16-битное произведение в регистре ax. Например, во фрагменте программы:

```
    ;
mov   al, 25
mov   dh, 40
mul   dh
```

al умножается на dh, а результат (1000) помещается в регистр ax. Заметим, что в инструкции mul требуется указывать только один операнд, другой сомножитель хранится в регистре al (в регистре ax в случае перемножения 16-битных сомножителей или в регистре eax — в случае 32-битных).

Инструкция перемножения 16-битных сомножителей работает аналогично.

Один из сомножителей должен храниться в регистре ax, а другой может находиться в 16-разрядном регистре общего назначения или в переменной памяти. 32-битовое произведение инструкция mul помещает в этом случае в регистровую пару dx:ax, при этом младшие (менее значащие) 16 битов произведения записываются в регистр ax, а старшие (более значащие) 16 бит — в регистр dx. Например, инструкции:

```
    ;
mov   ax, 1000
mul   ax
```

загружают в регистр ax значение 1000, а затем возводят его в квадрат, помещая результат (значение 1000000) в регистры dx:ax. Инструкция умножения 32-битных чисел аналогична таковой для 16-битных, с той разницей, что сомножитель находится в регистре eax, а 64-битовое произведение помещается в регистры edx:eax.

В отличие от сложения и вычитания, для работы со знаковыми операндами имеется вторая инструкция умножения imul для умножения 8-, 16- и 32-битных сомножителей. Если не принимать во внимание, что перемножаются значения со знаком, инструкция imul работает аналогично инструкции verb!.

Например, при выполнении инструкций:

```
    ;
mov   al, -2
mov   ah, 10
imul  ah
```

в регистре ax будет записано значение -20.

7.3 Задание для выполнения

Написать программу, реализующую функции простейшего калькулятора для работы с двумя целыми положительными числами в диапазоне 0...65635. Реализовать операции +, –, *, /.

Программа должна работать циклически.

При вводе недопустимого значения должно выдаваться диагностическое сообщение.

7.4 Контрольные вопросы

1. Как задается разрядность умножения?
2. Как работает инструкция перемножения 16-битных сомножителей?
3. Как проверить произошёл ли перенос в старшее слово при перемножении 16-битных сомножителей?

8 Изучение строковых инструкций. Работа с файлами

8.1 Краткие теоретические сведения

8.1.1 Строковый примитив поиска

Для поиска определенного символа или слова в строке используют примитив `scas`. Значение для поиска задается в регистре `eax`. Адрес строки определяется адресом в `edi`.

8.1.2 Префикс повторения строкового примитива

Для обработки всей строки необходимо применить примитив к каждому символу. Поскольку при поиске возможны 2 результата — найдено и не найдено, существует 2 префикса повторения этого примитива:

- `repe`, `repz` — повторять пока совпадает;
- `repne`, `repnz` — повторять пока не совпадает.

```
;
        mov     ecx, 100        ; размер строки
        mov     eax, 1234h
        mov     edi, Buff       ; ее адрес
        repne   scasw           ; ищем слово 1234h
```

Число повторений строкового примитива определяется значением в регистре `ecx` либо исходом поиска.

8.1.3 Строковый примитив сравнения строк

Для сравнения двух строк используют примитив `cmps`. При этом `esi` указывает на одну строку, а `edi` — на другую. Как и для `scas`, тут применяют два префикса повторения `repe` и `repne` — для первого несовпадения или совпадения соответственно.

```
;
        mov   ecx, 100
        mov   esi, Str1
        mov   edi, Str2
        repe  cmpsb     ; сравнивать, пока совпадают
        jne   NotEq     ; различны
```

Примитив `cmps` устанавливает те же флаги, что и инструкция `cmp`. Поэтому строки можно сравнивать не только на равенство или неравенство. Но при необходимости более сложного сравнения нужно использовать циклы.

```
;
        mov   ecx, 100
        mov   esi, Str1
        mov   edi, Str2
@@1:    cmpsb
```

```
jg     S1GT        ; байт si-строки больше байта di-строки
loop @@1
```

8.1.4 Работа с файлами

Для открытия существующего файла или создания нового используют функцию `open` (системный вызов номер 5). В синтаксисе Си она выглядит следующим образом:

```
int open (const char *filename, int flags[, mode_t mode])
```

Функция `open` создает и возвращает новый дескриптор для указанного файла. Индикатор текущей позиции при этом находится в начале файла. Функция может иметь переменный набор аргументов; аргумент `mode` используется только при создании файла и задает права доступа к нему (в стандартном для UNIX-систем числовом виде, например восьмеричным числом из трех цифр).

Первый аргумент задает имя открываемого файла — полный путь к файлу от корня файловой системы, либо относительный от текущего каталога.

Примечание: Имя файла должно быть задано в виде классической ASCIIZ-строки. Поэтому при вводе имени файла с клавиатуры необходимо обеспечить, чтобы в конце введенной строки находится нуль-терминатор, а не символ `\n`.

Аргумент `flags` задает режим открытия файла. Это — битовая маска; вы создаете значение поразрядным ИЛИ соответствующих параметров. Аргумент `flags` должен обязательно включать одно из этих значений, для задания режима доступа к файлу:

`O_RDONLY=0`	Открывает файл для чтения.
`O_WRONLY=1`	Открывает файл для записи.
`O_RDWR=2`	Открывает файл и для чтения и для записи.

Аргумент `flags` может также включать любую комбинацию следующих флагов:

O_APPEND=2000	Если установлен, то все операции записи запишут данные в конец файла, расширяя его, независимо от текущей файловой позиции.
O_CREAT=100	Если установлен, будет создан файл, если он еще не существует.
O_EXCL=200	Если и O_CREAT и O_EXCL установлены, то open выдает ошибку, если заданный файл уже существует.
O_NOCTTY=400	Если filename — имя терминала, не делайте его терминалом управления для процесса.
O_NONBLOCK=4000	Устанавливает режим неблокирования. Эта опция обычно полезна для специальных файлов FIFO и устройств типа терминалов. Обычно, для этих файлов open блокируется, пока файл не готов. Если O_NONBLOCK установлен, open возвращается немедленно. O_NONBLOCK-бит также воздействует на чтение и на запись: он разрешает им возвращаться немедленно с состоянием ошибки, если не имеется никакого доступного ввода, или если вывод не может быть записан.
O_TRUNC=1000	Если файл существует и открыт для записи, он усекается до нулевой длины. Эта опция полезна только для регулярных файлов, а не специальных файлов типа каталогов или FIFO.

Файл /usr/include/fcntl.h содержит более подробную информацию о значениях флагов.

Нормальное возвращаемое значение open — неотрицательное целое число, равное дескриптору файла. В случае ошибки возвращается значение -1.

Закрытие файла выполняется функцией close (системный вызов номер 6):

```
int close (int filedes)
```

Функция закрывает дескриптор файла filedes. Закрытие файла имеет следующие последствия:

- Описатель файла освобожден.
- Любые блокировки записи, принадлежащие процессу на файле, разблокирутся.

Нормальное возвращаемое значение — 0; значение -1 возвращается в случае ошибки.

Для получения или изменения текущего значения файловой позиции по дескриптору файла используют функцию lseek (системный вызов 19):

```
off_t lseek (int filedes, off_t offset, int whence)
```

Функция lseek используется, чтобы изменить файловую позицию файла с описателем filedes. Аргумент offset — новое смещение файловой позиции относительно whence. Аргумент whence может быть одной из символических констант:

SEEK_SET=0 Определяет, что whence — число символов от начала файла.

SEEK_CUR=1 Определяет, что whence — число символов от текущей файловой позиции. Этот число может быть положительно или отрицательно.

SEEK_END=2 Определяет, что whence — число символов с конца файла. Отрицательное число определяет позицию внутри текущего тела файла; положительное число определяет позицию после текущего конца. Если вы устанавливаете позицию после текущего конца, и фактически записываете данные, вы расширяете файл нулями до этой позиции.

Возвращаемое значение lseek — обычно возникающая в результате файловая позиция, измеряемая в байтах от начала файла. Вы можете использовать это средство вместе с SEEK_CUR для чтения текущей файловой позиции.

Вы можете устанавливать файловую позицию после текущего конца файла. Результатом будет соответствующее увеличение размера файла. Сама функция lseek не изменяет файл, но последующий вывод в новую позицию расширит файл.

Если файловая позиция не может быть изменена, или операция выполняется некоторым недопустимым способом, lseek возвращает значение -1.

Можно использовать lseek (desc, 0, SEEK_CUR) для получения текущего значения файловой позиции из описателя, а также lseek (desc, 0, SEEK_END) для определения размера файла.

8.2 Задание для выполнения

Написать программу со следующим алгоритмом:

- вывести на экран приглашение, и ввести с клавиатуры символьную подстроку для поиска;
- вывести на экран еще одно приглашение, и ввести с клавиатуры символьную строку с именем файла;
- открыть файл, прочитать его в буфер и найти в содержимом подстроку, используя строковые инструкции;
- вывести результат поиска в виде сообщения «Найдено» или «Не найдено»;
- завершить программу.

8.3 Контрольные вопросы

1. Каковы особенности строковых инструкций?
2. Для чего используется флаговый регистр в строковых инструкциях?
3. Каково назначение и механизм применения префикса повторения строкового примитива?
4. Что возвращается в случае ошибки открытия файла?

9 Управление отображением вывода в терминал ОС GNU/Linux

9.1 Краткие теоретические сведения

9.1.1 Системные инструменты

Представление информации в устройствах стандартного ввода-вывода

Для типового компьютера клавиатура является главным средством ввода информации, а дисплей — главным средством вывода. Знание особенностей представления информации в них является необходимым фактором умения работы с машиной. Различают три уровня представления и обработки сигналов, поступающих с клавиатуры: физический, логический и функциональный.

Физический уровень имеет дело с сигналами, которые поступают в системный блок при нажатии и отпускании клавиш. При нажатии любой клавиши в системный блок посылается код, соответствующий ее порядковому номеру, который называют скан-кодом клавиши (от англ. scan-code). При отпускании клавиши также генерируется ее номер, увеличенный на 128 (дополнительный скан-код).

На *логическом* уровне, реализуемом базовой подсистемой ввода-вывода ОС, происходит трансляция поступающего с клавиатуры скан-кода для отображения на дисплее соответствующего символа.

На *функциональном* уровне отдельным клавишам приписываются определенные функции, которые реализуются при нажатии этих клавиш.

Во времена создания UNIX для работы с ЭВМ использовались телетайпные терминалы. Терминал находился далеко от компьютера и был связан с ним кабелем по последовательному интерфейсу. Отображение информации на терминале можно было настроить с помощью посылки набора байтов к каждому из них. Всеми возможностями терминалов можно управлять специальными escape-последовательностями — особыми наборами байтов, которые начинаются с символа escape («эскейп», сокращенно «ESC», код 0x1B). В наши дни, работая с программами эмуляции терминала (для краткости их обычно называют просто «терминалами»), мы можем послать на стандартный вывод escape-последовательность, и это будет иметь тот же эффект, что и на аппаратном терминале.

Наберите на вашей консоли следующее:

```
echo "^[[0;31;40mIn Color"
```

Первый символ является символом escape, и выглядит как два символа: «^» и «[». Что бы ввести этот символ, вам необходимо на-

жать CTRL+V и потом клавишу ESC (впрочем, сочетание клавиш может зависеть от программы-терминала). Все остальные символы являются обычными печатными. Вы увидите строку «In Color» красного цвета. Консоль так и останется в режиме красного цвета, и чтобы вернуть обычное состояние, надо ввести теперь:

```
echo "^[[0;37;40m"
```

Как можно видеть, достаточно просто устанавливать и сбрасывать цвет. Существует большое число escape-последовательностей, с помощью которых вы можете выполнять множество вещей: перемещать курсор, сбрасывать терминал и т. п.

Последовательность для установки цветов

Последовательность, которая должна быть выведена на терминал для установки цветов, следующая:

```
<ESC>[{attr};{fg};{bg}m
```

Первый символ — это ESC, который вводится нажатием CTRL+V и ESC на Linux-консоли или же в xterm, konsole, kvt, и т.д. («CTRL+V ESC» можно применять и в текстовом редакторе vim для вставки символа ESC). Далее `{attr}`, `{fg}`, `{bg}` должны быть заменены верными значениями для получения соответствующего эффекта. `attr` — это атрибут, вроде мигания или подчёркивания. `fg` и `bg` — это цвета символов и фона соответственно. Вам не нужно брать номера в фигурные скобки. Просто напишите их, этого достаточно.

`{attr}` может принимать следующие значения:

0 сбросить все атрибуты (вернуться в нормальный режим)
1 яркий (обычно включает жирный шрифт)
2 тусклый
3 подчёркнутый
5 мигающий
7 инверсный
8 невидимый

`{fg}` может принимать следующие значения:

30 чёрный
31 красный
32 зелёный
33 жёлтый
34 синий
35 фиолетовый
36 голубой
37 белый

`{bg}` может принимать следующие значения:

40 чёрный
41 красный
42 зелёный
43 жёлтый
44 синий
45 фиолетовый
46 голубой
47 белый

Так, для получения мигающего синего текста на зелёном фоне нужно вывести комбинацию `echo "^[[5;34;42mIn color"`, а вернуть все назад можно комбинацией `echo "^[0;37;40m"`.

Команды (esc-последовательности) syscons

Далее во всех командах числовой аргумент обозначается буквой `n` или `n1,n2` и т. д., если их может быть несколько, `Esc` обозначает символ escape, а все остальные буквы являются частью команды.

При этом если команда требует числовой аргумент (или несколько), его можно пропустить. В этом случае обычно подразумевается, что он равен 1.

`Esc7` или `Esc[s`	запомнить положение курсора
`Esc8` или `Esc[u`	восстановить запомненное положение курсора
`Escc`	очистить экран и установить курсор в левый верхний уг

Перемещение курсора:

`Esc[nA`	вверх на n строк
`Esc[nB` или `Esc[ne`	вниз на n строк
`Esc[nC` или `Esc[na`	вправо на n позиций
`Esc[nD`	влево на n позиций
`Esc[nE`	в начало строки и на n строк вниз
`Esc[nF`	в начало строки и на n строк вверх
`Esc[n1;n2f` или `Esc[n1;n2H`	переместить в позицию n1 и строку n2
`Esc[nZ`	на n табуляций назад (как Tab, но в обратную сторону)
`Esc[n‘`	в той же строке в позицию n
`Esc[nd`	в той же позиции в строку n
`EscM`	сдвинуть курсор на строчку вверх, если он был в самой верхней строке, то сдвинуть содержимое экрана на строчку вниз (то же самое, что делает NewLine, только «вверх ногами»)

Очистка части экрана:

`Esc[0J` от курсора до конца экрана
`Esc[1J` от начала экрана до курсора
`Esc[2J` весь экран
`Esc[0K` от курсора до конца строки
`Esc[1K` от начала строки до курсора
`Esc[2K` всю строку
`Esc[nX` очистить n знаков от позиции курсора

«Раздвижка», «сдвижка», «прокрутка»:

`Esc[nL` вставить n пустых строк (те, что были раздвинуть)
`Esc[nM` удалить n строк (те, что остались уплотнить)
`Esc[nP` удалить n знаков в строке (те, что остались уплотнить)
`Esc[n@` вставить n знаков в строку (те, что были уплотнить)
`Esc[nS` прокрутить содержимое экрана на n строк вверх
`Esc[nT` прокрутить содержимое экрана на n строк вниз

Другие команды:

`Esc[nz` переключится в виртуальный терминал n

`Esc[=n1;n2B` установить параметры встроенного динамика — «пищалки»
`n1` = частота (точнее, делитель для частоты), `n2` = длительность (стандартные значения - 800;1); влияет на «писк», который получается при выводе символа `Bell` — 7; параметры свои для каждого виртуального терминала

`Esc[=nC` тип курсора (действует на все виртуальные терминалы)
в `n` младший бит определяет мерцание (1 — да, 0 — нет)
следующий бит — тип курсора (аппаратно-генерируемый — 0, «символьный» — 1)

`Esc[=n1;n2C` форма «символьного» курсора (для каждого виртуального терминала своя); закрашиваются строчки с `n1` по `n2` (начиная сверху) в матрице знака, которым рисуется курсор

9.1.2 Элементы программирования

Отображение файла в память

Помимо стандартного чтения файла в буфер заданными порциями байт в Linux существует еще как минимум один способ получить доступ к его содержимому: отображение файла на область памяти (функция `mmap`).

Механизм его работы следующий. Как только происходит обращение к памяти по указателю, который возвратила функция mmap, автоматически загружаются данные с диска в файловый кэш (если они еще не в кэше) и делается отображение (mapping) кэша на адресное пространство программы, после чего программе дается право на чтение этих данных.

Это позволяет программисту вообще не заботиться об оптимизации работы с диском — все это берет на себя механизм виртуальной памяти Linux. В любом случае происходит экономия и памяти, и скорости (за счет отсутствия копирования из кэша в буфер приложения). Это особенно заметно в случае обращения к одним и тем же данным несколькими приложениями.

Для работы с отображением файлов в память используются функции mmap (отображение памяти, системный вызов 90) и munmap (отмена отображения памяти, системный вызов 91):

```
void * mmap(void *start, size_t length, int prot , int flags, i
           off_t offset);
int  munmap(void *start, size_t length);
```

Функция mmap отражает length байтов, начиная со смещения offset файла (или другого объекта), определенного файловым описателем fd, в память, начиная с адреса start. Последний параметр (адрес) необязателен, и обычно бывает равен 0. Настоящее местоположение отраженных данных возвращается самой функцией mmap, и никогда не бывает равным 0. Аргумент prot описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла). Оно является либо PROT_NONE, либо побитовым ИЛИ одного или нескольких флагов PROT_*:

PROT_NONE=0	доступ к этой области памяти запрещен.
PROT_READ=1	данные можно читать;
PROT_WRITE=2	в эту область можно записывать информацию;
PROT_EXEC=4	данные в страницах могут исполняться;

Параметр flags задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_SHARED=1 — разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл.

MAP_PRIVATE=2 — создать неразделяемое отражение с механизмом copy-on-write. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова mmap видимыми в отраженном диапазоне.

`fd` должно быть корректным описателем файла. `offset` должен быть пропорционален размеру страницы виртуальной памяти (например 0).

При удачном выполнении `mmap` возвращает указатель на область с отраженными данными. При ошибке возвращается значение `MAP_FAILED` (-1).

9.2 Задание для выполнения

Написать программу со следующим алгоритмом:

- получить из стека командную строку (содержащую имя файла и параметры, переданные программе при запуске);
- первый параметр вывести на экран красным цветом, второй — синим и с новой строки.
- выполнить побайтный «переворот» содержимого файла, имя которого задано первым параметром командной строки; информацию записывать в этот же файл без промежуточных и дополнительных файлов, с использованием отображения файла на память;
- каждая файловая операция контролируются на ошибку без анализа ее номера с выдачей на терминал своего сообщения, типа «Ошибка открытия файла», «Ошибка записи» и т.д.
- завершить программу.

9.3 Контрольные вопросы

1. Как можно устанавливать и сбрасывать цвет на терминале?
2. Как определить произошла ли ошибка записи в файл?
3. Что возвращает системный вызов mmap при удачном выполнении отображения данных?

10 Работа с директориями в ОС GNU/Linux

10.1 Краткие теоретические сведения

10.1.1 Получение информации о содержимом текущего каталога

Типичная файловая система в GNU/Linux (и в других Unix-подобных ОС) содержит объекты трех типов.

- *Блок данных*. Это адресуемый на диске фрагмент файла. Зная, где находятся все блоки данных файла и, получив к ним доступ, мы, тем самым, получаем доступ к содержимому файла. Изменяя блоки данных, мы меняем содержимое файла.
- *Индексный дескриптор (inode)*. Этот объект хранит метаданные файла. В нем в том числе содержится информация о правах на файл и месте расположения на диске блоков данных файла. Но в индексном дескрипторе не содержится имени файла.
- *Директория или каталог*. Директория — это особый вид файла, который содержит сопоставление между именами файлов, которые «хранятся в этой директории», и индексными дескрипторами, соответствующими этим файлам. Конечно, как и у каждого файла, у директории есть свой индексный дескриптор, иначе нельзя было бы узнать, «внутри» какой директории «лежит» данная директория.

Хотя директории, по-существу, сами являются специализированными файлами, формат представления данных в них различается для разных файловых систем. Поэтому для получения списка файлов, находящихся в директории, программе необходим способ, не зависящий от типа файловой системы. Для получения списка файлов в таком универсальном формате существует системный вызов `getdents` с номером 141:

```
int getdents(unsigned int fd, struct dirent *dirp,
             unsigned int count);
```

В качестве параметров этому системному вызову передаются:

- `unsigned int fd` — файловый дескриптор данной директории (полученный, например, системным вызовом `open`);
- `struct dirent *dirp` — адрес буфера в памяти, куда будет записана информация о содержимом текущего каталога в виде следующих друг за другом структур;
- `unsigned int count` — размер буфера, в который должна быть записана информация.

Возвращаемое этим системным вызовом значение — неотрицательное целое число, равное количеству фактически прочитанных байт.

Системный вызов записывает информацию о файле в виде структурной переменной типа `dirent`. Эта структурная переменная содержит информацию, не зависящую от типа файловой системы. Структура содержит следующие поля:

```
    ;
struct dirent {
      long d_ino; /* номер inode */
      off_t d_off; /* смещение следующего элемента dirent в каталоге*/
      unsigned short d_reclen; /* длина данного dirent */
      char d_name []; /* имя файла в ASCIIZ-формате */
}
```

`d_ino` — это inode файла, `d_off` — смещение в каталоге в реальной файловой системе (смысл значения `d_off` в некоторых файловых системах оказывается не вполне тривиальным, поэтому данное поле не рекомендуется к использованию напрямую в прикладных программах). Массив `d_name` содержит строку с именем файла, и имеет разную длину для разных файлов. Поэтому структурные переменные типа `dirent` имеют разный размер, и размер данной конкретной структуры хранится в поле `d_reclen`. Эта длина определяется как число байт между текущим элементом и следующим, причем следующий элемент всегда будет выравнен по границе значения типа `long`.

Хотя в языке ассемблера нет специального типа данных, соответствующего структурам языка Си, к полям структурной переменной можно получить доступ, зная их размер. На языке ассемблера для этого к базовому адресу структурной переменной добавляется смещение, по которому находится нужное поле от начала структурной переменной.

В Си компилятор руководствуется следующими правилами при расположении полей внутри структурной переменной:

- Вся структура должна быть выровнена в памяти так, как выровнен её элемент с наибольшим выравниванием.
- Каждый элемент находится по наименьшему следующему адресу с подходящим выравниванием. Если необходимо, для этого между полями структуры добавляется нужное число байт-заполнителей. Размер структуры должен быть кратен её выравниванию. Если необходимо, для этого в конец структуры включается нужное число байт-заполнителей.

В нашем случае наибольшее по размеру поле — `d_ino`. Это число размером 8 байт (для 32-битных систем). В результате, все смещения полей структурной переменной `dirent` окажутся кратны 8 байтам.

Примечание: можно воспользоваться альтернативным способом определения смещений полей в структуре. Для этого нужно написать простую программу на Си, выводящую на экран соответствующие размеры, выполнить ее и посмотреть результаты эксперимента. Этот же способ можно использовать, чтобы выяснить размер неизвестного типа

```
struct                  Выравнивание структуры: 1, размер: 1
{
  char c;               | c |
};
                        0

struct                  Выравнивание структуры: 2, размер: 4
{
  char c;               | c | XX |    s    |
  short s;
};                      0       2

struct                  Выравнивание структуры: 4, размер: 8
{
  char c;               | c | XX | XX | XX |        i        |
  int i;
};                      0                   4

struct                  Выравнивание структуры: 4, размер: 8
{
  int i;                |        i        | c | XX  XX  XX |
  char c;
};                      0                   4
```

данных: например узнать размер типа `off_t` можно, выполнив в Си-программе инструкцию `printf("%d", sizeof(off_t))`.

В структуре `dirent` для нас представляет интерес поле `d_name`, содержащее имя файла, и поле `d_reclen` (беззнаковое целое размером 2 байта в 32-битной системе). Поскольку значение `d_reclen` — это размер текущей структурной переменной, оно равно числу байт между полем текущей структурной переменной и одноименным полем следующей структурной переменной в буфере.

10.2 Задание для выполнения

Составить алгоритм программы, выполняющей побайтный «переворот» содержимого всех файлов, лежащих в текущей директории.

- файл длины не более 60 Kb;
- информация записывается в новый файл, в конце имени которого содержится знак тильда («~»); любые промежуточные и дополнительные файлы отсутствуют;
- каждая файловая операция контролируется на ошибку с выдачей на терминал соответствующего сообщения, типа «Ошибка открытия файла», «Ошибка записи» и т.д.;
- применить буферизацию с большими (десятки килобайт) буферами.

Примерная последовательность действий может быть следующей:

1. получить системным вызовом `open` файловый дескриптор данной директории;

2. получить список файлов, находящихся в директории с помощью системного вызова `getdents` (с номером 141 в 32-битной системе);
3. Из структурной переменной типа `dirent` получить имя файла; открыть его для чтения, используя системный вызов `open` (с номером 5 в 32-битной системе). Эта операция проверяет правильность имени файла и его наличие на диске. Если файл отсутствует, то операция возвращает в регистре `eax` отрицательное значение. Не забывайте проверять это;
4. определить длину файла. Для определения длины файла можно использовать, например, способ, описанный в работе № 8 (с помощью системного вызова `lseek` получить значение позиции в файле, соответствующей смещению от конца файла);
5. создать новый файл с символом «~» в конце имени;
6. читать из исходного файла блок размером 60 Kb в буфер;
7. выполнить побайтный «переворот» буфера;
8. записать 60 Kb из буфера в новый файл (с текущей позиции, т. е. без перемещения файлового указателя);
9. проверить, прочитан ли весь исходный файл, и если нет то продолжить с пункта 6, иначе приступить к перевороту следующего файла в директории и вернуться к пункту 3.

10.3 Контрольные вопросы

1. Объекты каких типов содержит Типичная файловая система в GNU/Linux?
2. Как можно получить список файлов находящихся в директории, для разных файловых систем?

Литература

1. Зубков С. В. Assembler для DOS, Windows и UNIX. — М.: ДМК пресс, 2013. — 638 с.
2. Кип И. Язык Ассемблера для процессоров Intel. — М.: Вильямс, 2005. — 912 с.
3. Магда Ю. Ассемблер для процессоров Intel Pentium. — СПб.: Питер, 2006. — 410 с.
4. Столяров А. В. Программирование на языке ассемблера NASM в ОС Unix: уч. пособие. — 2-е изд. — М.: МАКС пресс, 2011. — 188 с.
5. Леонов В. Команды Linux. — М.: ЭКСМО, 2011. —176 с.
6. Роббинс А. Программирование в примерах. —М.: Кудиц-образ, 2008. — 656 с.
7. Таненбаум Э., Остин Т. Архитектура компьютера — СПб.: Питер, 2013. — 816 с.

УЧЕБНОЕ ИЗДАНИЕ

Костюк Дмитрий Александрович
Четверкина Галина Андреевна

Программирование на ассемблере в GNU/Linux

МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальностей 1-40 01 02 и 1-36 04 02

Ответственный за выпуск: Д. А. Костюк
Редактор: Т. В. Строкач
Компьютерная верстка: Д. А. Костюк
Корректор: Е. В. Никитчик

Подписано к печати 04.03.2011 г. Бумага «Снегурочка». Формат 60x84 1/16. Гарнитура «Arial». Усл. п. л. 22,2. Уч. изд. л. 23,9. Тираж 30 экз. Заказ № 174. Отпечатано на ризографе учреждения образования «Брестский государственный технический университет».
224017, г. Брест, ул. Московская, 267.
Лицензия № 02330/0549453 от 08.04.2009 г.