Библиотека сайта rus-linux.net

Школа ассемблера: разработка операционной системы

Оригинал: AsmSchool: Make an operating system
Автор: Mike Saunders
Дата публикации: 15 апреля 2016 г.
Перевод: А. Панин
Дата перевода: 16 апреля 2016 г.

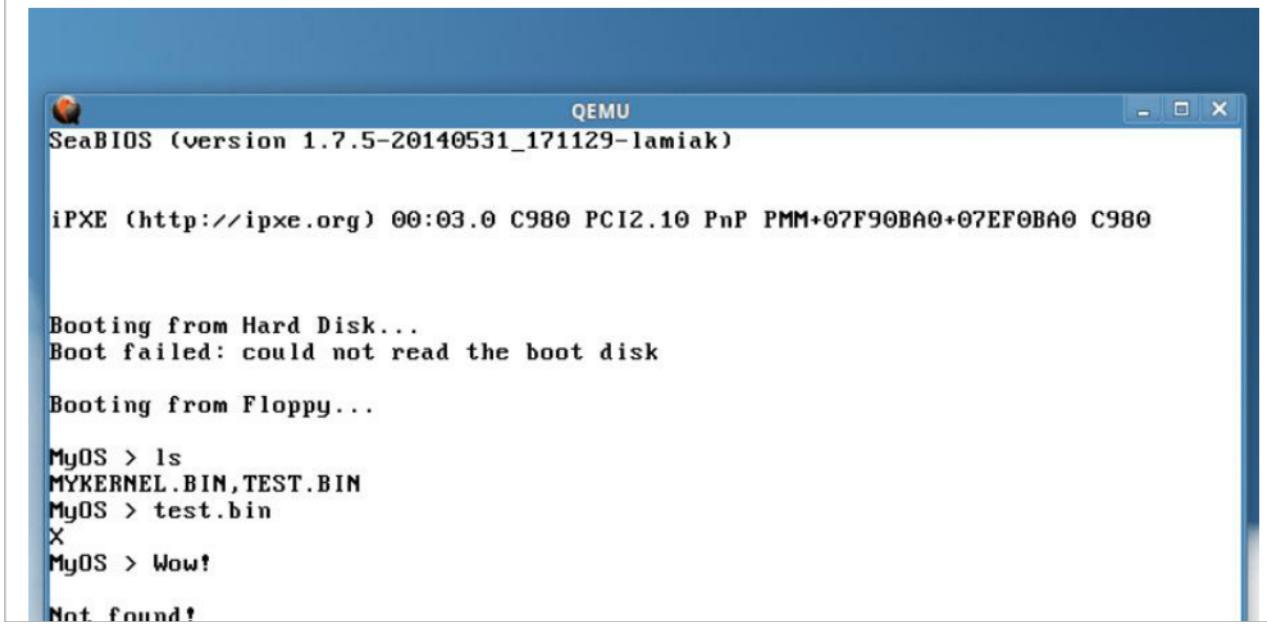
Часть 4: Располагая навыками, полученными в ходе чтения предыдущих статей серии, вы можете приступить к разработке своей собственной операционной системы!

Для чего это нужно?

- Для понимания принципов работы компиляторов.
- Для понимания инструкций центрального процессора.
- Для оптимизации вашего кода в плане производительности.

В течение нескольких месяцев мы прошли сложный путь, который начался с разработки простых программ на языке ассемблера для Linux и закончился в прошлом статье серии разработкой самодостаточного кода, исполняющегося на персональном компьютере без операционной системы. Ну а сейчас мы попытаемся собрать всю информацию воедино и создать самую настоящую операционную систему. Да, мы пойдем по стопам Линуса Торвальдса, но для начала стоит ответить на следующие вопросы: "Что же представляет собой операционная система? Какие из ее функций нам придется воссоздать?".

В данной статье мы сфокусируемся лишь на основных функциях операционной системы: загрузке и исполнении программ. Сложные операционные системы выполняют гораздо большее количество функций, таких, как управление виртуальной памятью и обработка сетевых пакетов, но для их корректной реализации требуются годы непрерывной работы, поэтому в данной статье мы рассмотрим лишь основные функции, присутствующие в любой операционной системе. В прошлом месяце мы разработали небольшую программу, которая умещалась в 512-байтовом секторе флоппи-диска (его первом секторе), а сейчас мы немного доработаем ее с целью добавления функции загрузки дополнительных данных с диска.



Наша операционная система в работе: вывод приветствия, исполнение команды и запуск программы с диска

Разработка системного загрузчика

Мы могли бы попытаться максимально сократить объем бинарного кода нашей операционной системы с целью его размещения в первом 512-байтовом секторе флоппи-диска, том самом, который загружается средствами BIOS, но в таком случае у нас не будет возможности реализовать какие-либо интересные функции. Поэтому мы будем использовать эти 512 байт для размещения бинарного кода простого системного загрузчика, который будет загружать бинарный код ядра ОС в оперативную память и исполнять его. (После этого мы разработаем само ядро ОС, которое будет загружать бинарный код других программ с диска и также исполнять его, но об этом будет сказано чуть позже.)

Вы можете загрузить исходный код рассмотренных в статье примеров по ссылке www.linuxvoice.com/code/lv015/asmschool.zip. А это код нашего системного загрузчика из файла с именем `boot.asm`:

```
BITS 16
jmp short start ; Переход к метке с пропуском описания диска
nop ; Дополнение перед описанием диска
%include "bpb.asm"
start:
mov ax, 07C0h ; Адрес загрузки
mov ds, ax ; Сегмент данных
mov ax, 9000h ; Подготовка стека
mov ss, ax
mov sp, 0FFFFh ; Стек растет вниз!
cld ; Установка флага направления
mov si, kern_filename
call load_file
jmp 2000h:0000h ; Переход к загруженному из файла бинарному коду ядра ОС
kern_filename db "MYKERNELBIN"
%include "disk.asm"
times 510-($-$$) db 0 ; Дополнение бинарного кода нулями до 510 байт
dw 0AA55h ; Метка окончания бинарного кода системного загрузчика
buffer: ; Начало буфера для содержимого диска
```

В данном коде первой инструкцией центрального процессора является инструкция `jmp`, которая расположена после директивы `BITS`, сообщающей ассемблеру NASM о том, что используется 16-битный режим. Как вы наверняка помните из предыдущей статьи серии, исполнение загружаемого средствами BIOS с диска 512-байтного бинарного кода начинается с самого начала, но нам приходится осуществлять переход к метке для пропуска специального набора данных. Очевидно, что в прошлом месяце мы просто записывали код в начало диска (с помощью утилиты `dd`), а остальное пространство диска оставляли пустым.

Сейчас же нам придется использовать флоппи-диск с подходящей файловой системой MS-DOS (FAT12), а для того, чтобы корректно работать с данной файловой системой, нужно добавить набор специальных данных рядом с началом сектора. Этот набор называется "блоком параметров BIOS" (BIOS Parameter

Block - BPB) и содержит такие данные, как метка диска, количество секторов и так далее. Он не должен интересовать нас на данном этапе, так как подобным темам можно посвятить не одну серию статей, именно поэтому мы разместили все связанные с ним инструкции и данные в отдельном файле исходного кода с именем `bpb.asm`.

Исходя из вышесказанного, данная директива из нашего кода крайне важна:

```
%include "bpb.asm"
```

Это директива NASM, позволяющая включить содержимое указанного файла исходного кода в текущий файл исходного кода в процессе ассемблирования. Таким образом мы сможем сделать код нашего системного загрузчика максимально коротким и понятным, вынеся все подробности реализации блока параметров BIOS в отдельный файл. Блок параметров BIOS должен располагаться через три байта после начала сектора, а так как инструкция `jmp` занимает лишь два байта, нам приходится использовать инструкцию `nop` (ее название расшифровывается как "no operation" - это инструкция, которая не делает ничего, кроме траты циклов центрального процессора) с целью заполнения оставшегося байта.

Ничто не сравнится с наблюдением за собственноручно созданным программным продуктом, исполняющемся на реальном компьютере (а также за собственным отражением) - это просто круто!

Работа со стеком

Далее нам придется использовать инструкции, аналогичные рассмотренным в прошлой статье, для подготовки регистров и стека, а также инструкцию `cld` (расшифровывается как "clear direction"), позволяющую установить флаг направления для определенных инструкций, таких, как инструкция `lodsb`, которая после ее исполнения будет увеличивать значение в регистре `SI`, а не уменьшать его.

После этого мы помещаем адрес строки в регистр `SI` и вызываем нашу функцию `load_file`. Но задумайтесь на минуту - мы ведь еще не разработали эту функцию! Да, это правда, но ее реализацию можно найти в другом подключаемом нами файле исходного кода с именем `disk.asm`.

Файловая система FAT12, используемая на флоппи-дисках, которые форматируются в MS-DOS, является одной простейших существующих файловых систем, но для работы с ее содержимым также требуется немалый объем кода. Подпрограмма `load_file` имеет длину около 200 строк и не будет приведена в данной статье, так как мы рассматриваем процесс разработки операционной системы, а не драйвера для определенной файловой системы, следовательно, не очень разумно тратить таким образом место на страницах журнала. В общем, мы подключили файл исходного кода `disk.asm` практически перед окончанием текущего файла исходного кода и можем забыть про него. (Если же вас все-таки заинтересовала структура файловой системы FAT12, вы можете ознакомиться с отличным обзором по адресу http://tinyurl.com/fat12spec, после чего заглянуть в файл исходного кода `disk.asm` - код, содержащийся в нем, хорошо прокомментирован.)

В любом случае, подпрограмма `load_file` загружает бинарный код из файла с именем, заданном в регистре `SI`, в сегмент 2000 со сдвигом 0, после чего мы осуществляем переход к его началу для исполнения. И это все - ядро операционной системы загружено и системный загрузчик выполнил свою задачу!

Вы наверняка заметили, что в качестве имени файла ядра операционной системы в нашем коде используется `MYKERNELBIN` вместо `MYKERNEL.BIN`, которое вполне вписывается в схему имен 8+3, используемую на флоппи-дисках в DOS. На самом деле, в файловой системе FAT12 используется внутреннее представление имен файлов, а мы экономим место, используя имя файла, которое гарантированно не потребует реализации в рамках нашей подпрограммы `load_file` механизма поиска символа точки и преобразования имени файла во внутреннее представление файловой системы.

После строки с директивой подключения файла исходного кода `disk.asm` расположены две строки, предназначенные для дополнения бинарного кода системного загрузчика нулями до 512 байт и включения метки окончания его бинарного кода (об этом говорилось в прошлой статье). Наконец, в самом конце кода расположена метка `"buffer"`, которая используется подпрограммой `load_file`. В общем, подпрограмме `load_file` требуется свободное пространство в оперативной памяти для выполнения некоторых промежуточных действий в процессе поиска файла на диске, а у нас есть достаточно свободного пространства после загрузки системного загрузчика, поэтому мы размещаем буфер именно здесь.

Для ассемблирования системного загрузчика следует использовать следующую команду:

```
nasm -f bin -o boot.bin boot.asm
```

Теперь нам нужно создать образ виртуального флоппи-диска в формате MS-DOS и добавить бинарный код нашего системного загрузчика в его первые 512 байт с помощью следующих команд:

```
mkdosfs -C floppy.img 1440
dd conv=notrunc if=boot.bin of=floppy.img
```

На этом процесс разработки системного загрузчика можно считать оконченным! Теперь у нас есть образ загрузочного флоппи-диска, который позволяет загрузить бинарный код ядра операционной системы из файла с именем `mykernel.bin` и исполнить его. Далее нас ждет более интересная часть работы - разработка самого ядра операционной системы

Ядро операционной системы

Мы хотим, чтобы наше ядро операционной системы выполняло множество важных задач: выводило приветствие, принимало ввод от пользователя, устанавливало, является ли ввод поддерживаемой командой, а также исполняло программы с диска после указания пользователем их имен. Это код ядра операционной системы из файла `mykernel.asm`:

операционной системы из файла mykernel.asm:

```
mov ax, 2000h
mov ds, ax
mov es, ax
loop:
mov si, prompt
call lib_print_string
mov si, user_input
call lib_input_string
cmp byte [si], 0
je loop
cmp word [si], "ls"
je list_files
mov ax, si
mov cx, 32768
call lib_load_file
jc load_fail
call 32768
jmp loop
load_fail:
mov si, load_fail_msg
call lib_print_string
jmp loop
list_files:
mov si, file_list
call lib_get_file_list
call lib_print_string
jmp loop
prompt db 13, 10, "MyOS > ", 0
load_fail_msg db 13, 10, "Not found!", 0
user_input times 256 db 0
file_list times 1024 db 0
%include "lib.asm"
```

Перед рассмотрением кода следует обратить внимание на последнюю строку с директивой подключения файла исходного кода `lib.asm`, который также находится в архиве `asmschool.zip` с нашего веб-сайта. Это библиотека полезных подпрограмм для работы с экраном, клавиатурой, строками и дисками, которые вы также можете использовать - в данном случае мы подключаем этот файл исходного кода в самом конце основного файла исходного кода ядра операционной системы для того, чтобы сделать последний максимально компактным и красивым. Обратитесь к разделу "Подпрограммы библиотеки lib.asm" для получения дополнительной информации обо всех доступных подпрограммах.

В первых трех строках кода ядра операционной системы мы осуществляем заполнение регистров сегментов данными для указания на сегмент 2000, в который была осуществлена загрузка бинарного кода. Это важно для гарантированной корректной работы таких инструкций, как `lodsb`, которые должны читать данные из текущего сегмента, а не из какого-либо другого. После этого мы не будем выполнять каких-либо дополнительных операций с сегментами; наша операционная система будет работать с 64 Кб оперативной памяти!

Далее в коде расположена метка, соответствующая началу цикла. В первую очередь мы используем одну из подпрограмм из библиотеки `lib.asm`, а именно `lib_print_string`, для вывода приветствия. Байты 13 и 10 перед строкой приветствия являются символами перехода на новую строку, благодаря которым приветствие будет выводиться не сразу же после вывода какой-либо программы, а всегда на новой строке.

После этого мы используем другую подпрограмму из библиотеки `lib.asm` под названием `lib_input_string`, которая принимает введенные пользователем с

помощью клавиатуры символы и сохраняет их в буфере, указатель на который находится в регистре SI. В нашем случае буфер объявляется ближе к концу кода ядра операционной системы следующим образом:

```
user_input times 256 db 0
```

Данное объявление позволяет создать буфер длиной в 256 символов, заполненный нулями - его длины должно быть достаточно для хранения команд такой простой операционной системы, как наша!

Далее мы выполняем проверку пользовательского ввода. Если первый байт буфера `user_input` является нулевым, то пользователь просто нажал клавишу Enter, не вводя какой-либо команды; не забывайте о том, что все строки оканчиваются нулевыми символами. Таким образом, в данном случае мы должны просто перейти к началу цикла и снова вывести приветствие. Однако, в том случае, если пользователь вводит какую-либо команду, нам придется сначала проверить, не ввел ли он команду `ls`. До текущего момента вы могли наблюдать в наших программах на языке ассемблера лишь сравнения отдельных байт, но не стоит забывать о том, что также имеется возможность осуществления сравнения двухбайтовых значений или машинных слов. В данном коде мы сравниваем первое машинное слово из буфера `user_input` с машинным словом, соответствующим строке `ls` и в том случае, если они идентичны, перемещаемся к расположенному ниже блоку кода. В рамках этого блока кода мы используем другую подпрограмму из библиотеки `lib.asm` для получения разделенного запятыми списка расположенных на диске файлов (для хранения которого должен использоваться буфер `file_list`), выводим этот список на экран и перемещаемся назад в цикл для обработки пользовательского ввода.

Исполнение сторонних программ

Если пользователь не вводит команду `ls`, мы предполагаем, что он ввел имя программы с диска, поэтому имеет смысл попытаться загрузить ее. Наша библиотека `lib.asm` содержит реализацию полезной подпрограммы `lib_load_file`, которая осуществляет разбор таблиц файловой системы FAT12 диска: она принимает указатель на начало строки с именем файла посредством регистра `AX`, а также значение смещения для загрузки бинарного кода из файла программы посредством регистра `CX`. Мы уже используем регистр `SI` для хранения указателя на строку с пользовательским вводом, поэтому мы копируем этот указатель в регистр `AX`, после чего помещаем значение 32768, используемое в качестве смещения для загрузки бинарного кода из файла программы, в регистр`CX`.

Но почему мы используем именно это значение в качестве смещения для загрузки бинарного кода из файла программы? Ну, это просто один из вариантов карты распределения памяти для нашей операционной системы. Из-за того, что мы работаем в одном сегменте размером в 64 Кб, а бинарный код нашего ядра загружен со смещением 0, нам приходится использовать первые 32 Кб памяти для данных ядра, а остальные 32 Кб - для данных загружаемых программ. Таким образом, смещение 32768 является серединой нашего сегмента и позволяет предоставить достаточный объем оперативной памяти как ядру операционной системы, так и загружаемым программам.

После этого подпрограмма `lib_load_file` выполняет крайне важную операцию: если она не может найти файл с заданным именем на диске или по какой-то причине не может считать его с диска, она просто завершает работу и устанавливает специальный флаг переноса (carry flag). Это флаг состояния центрального процессора, который устанавливается в процессе выполнения некоторых математических операций и в данный момент не должен нас интересовать, но при этом мы можем определять наличие этого флага для принятия быстрых решений. Если подпрограмма `lib_load_asm` устанавливает флаг переноса, мы задействуем инструкцию `jc` (переход при наличии флага переноса - jump if carry) для перехода к блоку кода, в рамках которого осуществляется вывод сообщения об ошибке и возврат в начало цикла обработки пользовательского ввода.

В том же случае, если флаг переноса не установлен, можно сделать вывод, что подпрограмма `lib_load_asm` успешно загрузила бинарный код из файла программы в оперативную память по адресу 32768. Все что нам нужно в этом случае - это инициировать исполнение бинарного кода, загруженного по этому адресу, то есть начать исполнение указанной пользователем программы! А после того, как в этой программе будет использована инструкция `ret` (для возврата в вызывающий код), мы должны будем просто вернуться в цикл обработки пользовательского ввода. Таким образом мы создали операционную систему: она состоит из простейших механизмов разбора команд и загрузки программ, реализованных в рамках примерно 40 строк ассемблерного кода, хотя и с большой помощью со стороны подпрограмм из библиотеки `lib.asm`.

Для ассемблирования кода ядра операционной системы следует использовать следующую команду:

```
nasm -f bin -o mykernel.bin mykernel.asm
```

После этого нам придется каким-то образом добавить файл `mykernel.bin` в файл образа флоппи-диска. Если вы знакомы с приемом монтирования образов дисков с помощью loopback-устройств, вы можете получить доступ к содержимому образа диска `floppy.img`, воспользовавшись им, но существует и более простой способ, заключающийся в использовании инструментария GNU Mtools (www.gnu.org/software/mtools). Это набор программ для работы с флоппи-дисками, на которых используются файловые системы MS-DOS/FAT12, доступный из репозиториев пакетов программного обеспечения всех популярных дистрибутивов Linux, поэтому вам придется лишь воспользоваться утилитой `apt-get`, `yum`, `pacman` или любой другой утилитой, используемой для установки пакетов программного обеспечения в вашем дистрибутиве.

После установки соответствующего пакета программного обеспечения для добавления файла `mykernel.bin` в файл образа диска `floppy.img` вам придется выполнить следующую команду:

```
mcopy -i floppy.img mykernel.bin ::/
```

Обратите внимание на забавные символы в конце команды: двоеточие, двоеточие и слэш. Теперь мы почти готовы запуску нашей операционной системы, но какой в этом смысл, пока для нее не существует приложений? Давайте исправим это недоразумение, разработав крайне простое приложение. Да, сейчас вы будете разрабатывать приложение для своей собственной операционной системы - просто представьте, насколько поднимется ваш авторитет в рядах гиков. Сохраните следующий код в файле с именем `test.asm`:

```
org 32768
mov ah, 0Eh
mov al, 'X'
int 10h
ret
```

Данный код просто использует функцию BIOS для вывода символа 'X' на экран, после чего возвращает управление вызвавшему его коду - в нашем случае этим кодом является код операционной системы. Строка `org`, с которой начинается исходный код приложения, является не инструкцией центрального процессора, а директивой ассемблера NASM, сообщающей ему о том, что бинарный код будет загружен в оперативную память со смещением 32768, следовательно, необходимо пересчитать все смещения с учетом данного обстоятельства.

Данный код также нуждается в ассемблировании, а получившийся в итоге бинарный файл - в добавлении в файл образа флоппи-диска:

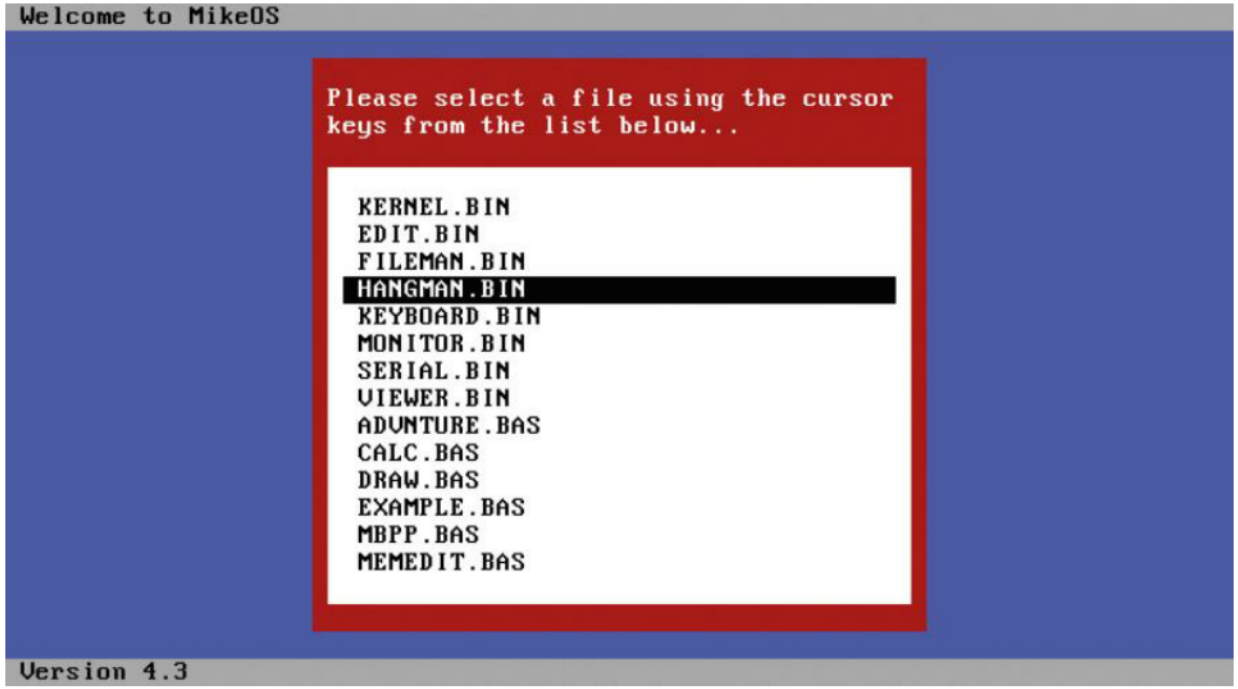
```
nasm -f bin -o test.bin test.asm
mcopy -i floppy.img test.bin ::/
```

Теперь глубоко вздохните, приготовьтесь к созерцанию непревзойденных результатов собственной работы и загрузите образ флоппи-диска с помощью эмулятора ПК, такого, как Qemu или VirtualBox. Например, для этой цели может использоваться следующая команда:

```
qemu-system-i386 -fda floppy.img
```

Вуаля: системный загрузчик `boot.img`, который мы интегрировали в первый сектор образа диска, загружает ядро операционной системы`mykernel.bin`, которое выводит приветствие. Введите команду `ls` для получения имен двух файлов, расположенных на диске (`mykernel.bin` и `test.bin`), после чего введите имя последнего файла для его исполнения и вывода символа X на экран.

Это круто, не правда ли? Теперь вы можете начать дорабатывать командную оболочку вашей операционной системы, добавлять реализации новых команд, а также добавлять файлы дополнительных программ на диск. Если вы желаете запустить данную операционную систему на реальном ПК, вам стоит обратиться к разделу "Запуск системного загрузчика на реальной аппаратной платформе" из предыдущей статьи серии - вам понадобятся точно такие же команды. В следующем месяце мы сделаем нашу операционную систему более мощной, позволив загружаемым программам использовать системные функции и реализовав таким образом концепцию разделения кода, направленную на сокращение его дублирования. Большая часть работы все еще впереди.



Наша операционная система является упрощенной версией операционной системы MikeOS (http://mikeos.sf.net), к исходному коду которой вы можете обращаться в поисках вдохновения

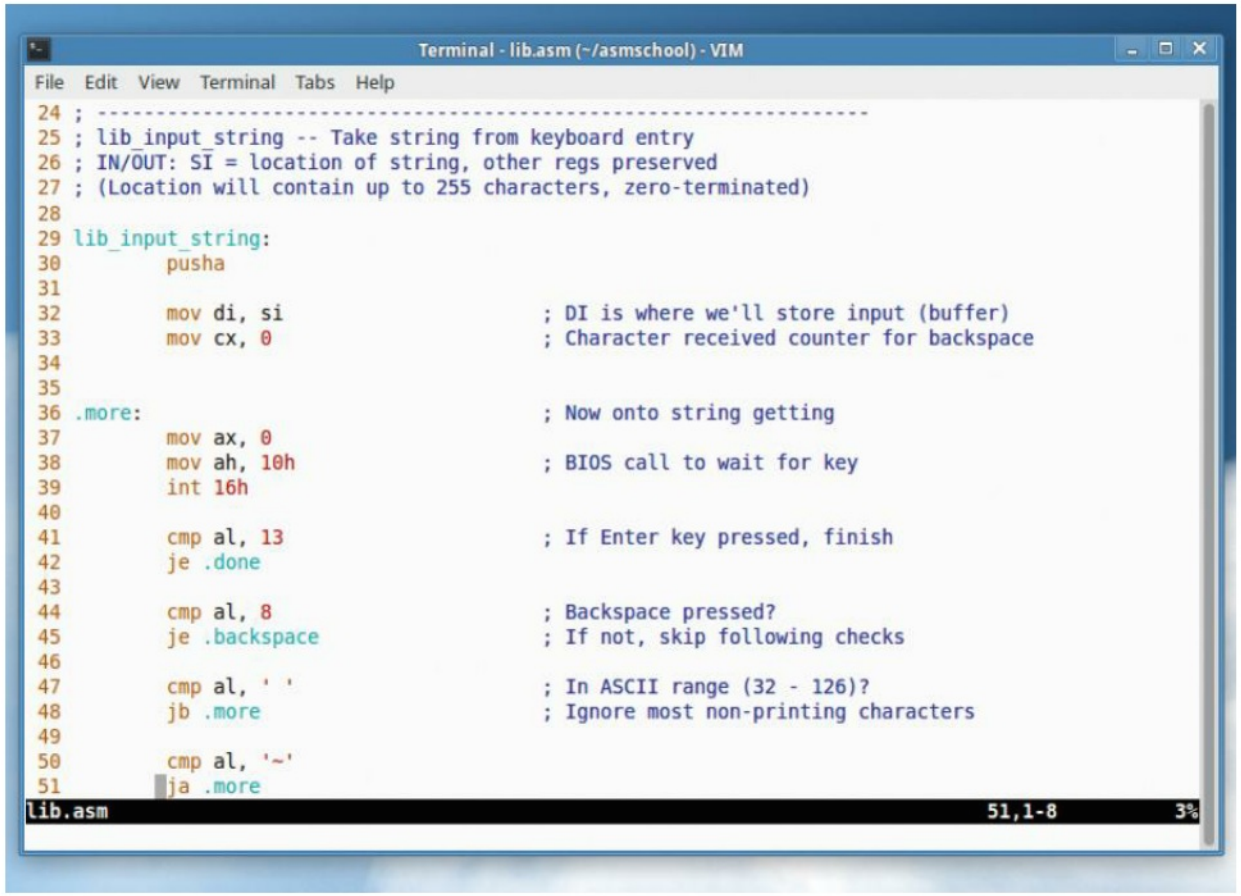
Подпрограммы библиотеки lib.asm

Как говорилось ранее, библиотека `lib.asm` предоставляет большой набор полезных подпрограмм для использования в рамках ваших ядер операционных систем и отдельных программ. Некоторые из них используют инструкции и концепции, которые пока не затрагивались в статьях данной серии, другие (такие, как подпрограммы для работы с дисками) тесно связаны с особенностями устройства файловых систем, но если вы считаете себя компетентным в данных вопросах, вы можете самостоятельно ознакомиться с их реализациями и разобраться в принципе работы. При этом более важно разобраться с тем, как вызывать их из собственного кода:

- `lib_print_string` - принимает указатель на завершающуюся нулевым символом строку посредством регистра `SI` и выводит эту строку на экран.
- `lib_input_string` - принимает указатель на буфер посредством регистра `SI` и заполняет этот буфер символами, введенными пользователем с помощью клавиатуры. После того, как пользователь нажимает клавишу Enter, строка в буфере завершается нулевым символом и управление возвращается коду вызывающей программы.
- `lib_move_cursor` - перемещает курсор на экране в позицию с координатами, передаваемыми посредством регистров `DH` (номер строки) и `DL` (номер столбца).
- `lib_get_cursor_pos` - следует вызывать данную подпрограмму для получения номеров текущей строки и столбца посредством регистров `DH` и `DL` соответственно.
- `lib_string_uppercase` - принимает указатель на начало завершающейся нулевым символом строки посредством регистра `AX` и переводит символы строки в верхний регистр.
- `lib_string_length` - принимает указатель на начало завершающейся нулевым символом строки посредством регистра `AX` и возвращает ее длину посредством регистра `AX`.

- `lib_string_compare` - принимает указатели на начала двух завершающихся нулевыми символами строк посредством регистров `SI` и `DI` и сравнивает эти строки. Устанавливает флаг переноса в том случае, если строки идентичны (для использования инструкции перехода в зависимости от флага переноса `jc`) или убирает этот флаг, если строки различаются (для использования инструкции `jnc`).
- `lib_get_file_list` - принимает указатель на начало буфера посредством регистра `SI` и помещает в этот буфер завершающуюся нулевым символом строку, содержащую разделенный запятыми список имен файлов с диска.
- `lib_load_file` - принимает указатель на начало строки, содержащей имя файла, посредством регистра `AX` и загружает содержимое файла по смещению, переданному посредством регистра `CX`. Возвращает количество скопированных в память байт (то есть, размер файла) посредством регистра `BX` или устанавливает флаг переноса, если файл с заданным именем не найден.

Попробуйте подключить код библиотеки `lib.asm` к коду ваших отдельных программ (таким же образом, как в файле `test.asm`) и протестируйте эти подпрограммы.



В библиотеке lib.asm полно полезных подпрограмм - внимательно присмотритесь к их реализациям

Предыдущие статьи из серии "Школа ассемблера":

- "Начинаем программировать на языке ассемблера: переход на уровень аппаратного обеспечения"

- "Школа ассемблера: условные инструкции, циклы и библиотеки"
- "Начинаем программировать на языке ассемблера"

Если вам понравилась статья, поделитесь ею с друзьями: