# Программирование на ассемблере x86

Unresolved directive in _index.adoc - include::shared/authors.adoc[] Unresolved directive in _index.adoc - include::shared/mirrors.adoc[] Unresolved directive in _index.adoc - include::shared/releases.adoc[] Unresolved directive in _index.adoc - include::shared/attributes/attributes-{{% lang %}}.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/teams.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/mailing-lists.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/urls.adoc[] toc::[]

*Эта глава была написана G. Adam Stanislav <adam@redprince.net>.*

# 1. Обзор

Программирование на ассемблере в UNIX® крайне плохо документировано. Обычно предполагается, что никто не захочет его использовать, поскольку различные системы UNIX® работают на разных микропроцессорах, и поэтому всё должно быть написано на C для обеспечения переносимости.

В действительности переносимость программ на C — это скорее миф. Даже программы на C требуют изменений при переносе с одной UNIX®-системы на другую, независимо от процессора, на котором они работают. Обычно такая программа содержит множество условных операторов, зависящих от системы, для которой она компилируется.

Даже если мы считаем, что всё программное обеспечение UNIX® должно быть написано на C или другом языке высокого уровня, нам всё равно нужны программисты на ассемблере: кто же ещё напишет часть библиотеки C, которая обращается к ядру?

В этой главе я попытаюсь показать вам, как можно использовать язык ассемблера для написания программ под UNIX®, в частности под FreeBSD.

В этой главе не объясняются основы языка ассемблера. Существует достаточно ресурсов на эту тему (например, полный онлайн-курс по языку ассемблера можно найти в Искусстве языка ассемблера Рэндалла Хайда; если вы предпочитаете печатные книги, обратите внимание на «Язык ассемблера шаг за шагом» Джеффа Дантемана (ISBN: 0471375233)). Однако после прочтения этой главы любой программист на языке ассемблера сможет писать программы для FreeBSD быстро и эффективно.

Copyright ® 2000-2001 G. Adam Stanislav. All rights reserved.

# 2. Инструменты

## 2.1. Ассемблер

Важнейшим инструментом для программирования на языке ассемблера является ассемблер — программа, преобразующая код на языке ассемблера в машинный код.

Три очень разных ассемблера доступны для FreeBSD. И llvm-as(1) (включён в devel/llvm), и as(1) (включён в devel/binutils) используют традиционный синтаксис ассемблера UNIX®.

С другой стороны, nasm(1) (устанавливаемый через devel/nasm) использует синтаксис Intel. Его основное преимущество в том, что он может ассемблировать код для многих операционных систем.

В этой главе используется синтаксис nasm, потому что большинство программистов на ассемблере, приходящих в FreeBSD из других операционных систем, найдут его более понятным. Кроме того, если честно, это то, к чему я привык.

## 2.2. Компоновщик

Результат работы ассемблера, как и любого компилятора, необходимо связать, чтобы получить исполняемый файл.

Стандартный компоновщик ld(1) поставляется с FreeBSD. Он работает с кодом, собранным любым из ассемблеров.

# 3. Системные вызовы

## 3.1. Стандартное соглашение о вызовах

По умолчанию ядро FreeBSD использует соглашение о вызовах C. Кроме того, хотя доступ к ядру осуществляется с помощью `int 80h`, предполагается, что программа вызовет функцию, которая выполняет `int 80h`, а не будет выполнять `int 80h` напрямую.

Эта традиция очень удобна и значительно превосходит соглашение Microsoft®, используемое в MS-DOS®. Почему? Потому что соглашение UNIX® позволяет любой программе, написанной на любом языке, обращаться к ядру.

Программа на ассемблере также может это сделать. Например, мы могли бы открыть файл:

```
kernel:
    int 80h ; Call kernel
    ret

open:
    push    dword mode
```

```
    push    dword flags
    push    dword path
    mov eax, 5
    call    kernel
    add esp, byte 12
    ret
```

Это очень понятный и переносимый способ написания кода. Если вам нужно перенести код на UNIX®-систему, которая использует другое прерывание или другой способ передачи параметров, все, что вам нужно изменить, это процедуру kernel.

Но программисты на ассемблере любят экономить такты. Приведённый выше пример требует комбинации `call/ret`. Мы можем исключить её, сделав `push` дополнительного двойного слова:

```
open:
    push    dword mode
    push    dword flags
    push    dword path
    mov eax, 5
    push    eax     ; Or any other dword
    int 80h
    add esp, byte 16
```

Помещённое в `EAX` значение `5` идентифицирует функцию ядра, в данном случае `open`.

## 3.2. Альтернативное соглашение о вызовах

FreeBSD — это чрезвычайно гибкая система. Она предлагает другие способы вызова ядра. Однако для работы необходимо, чтобы в системе была установлена эмуляция Linux.

Linux — это система, подобная UNIX®. Однако её ядро использует то же соглашение о системных вызовов для передачи параметров в регистрах, что и MS-DOS®. Как и в соглашении UNIX®, номер функции помещается в `EAX`. Однако параметры передаются не в стеке, а в регистрах `EBX, ECX, EDX, ESI, EDI, EBP`:

```
open:
    mov eax, 5
    mov ebx, path
    mov ecx, flags
    mov edx, mode
    int 80h
```

Этот подход имеет значительный недостаток по сравнению с UNIX®, по крайней мере, в контексте программирования на ассемблере: каждый раз при вызове ядра необходимо сохранять регистры с помощью `push`, а затем восстанавливать их с помощью `pop`. Это делает ваш код более громоздким и медленным. Тем не менее, FreeBSD предоставляет вам выбор.

Если вы решите использовать соглашение Linux, вы должны сообщить об этом системе. После того как ваша программа будет ассемблирована и слинкована, вам нужно пометить исполняемый файл:

```
% brandelf -t Linux filename
```

## 3.3. Какое соглашение следует использовать?

Если вы разрабатываете код специально для FreeBSD, всегда следует использовать соглашение UNIX®: это быстрее, вы можете хранить глобальные переменные в регистрах, вам не нужно маркировать исполняемый файл, и вы не требуете установки пакета эмуляции Linux на целевой системе.

Хотя вы можете хотеть создать переносимый код, который также работает на Linux, вам, вероятно, по-прежнему будет нужен максимально эффективный код для пользователей FreeBSD. Я покажу вам, как этого добиться, после того как объясню основы.

## 3.4. Номера вызовов

Чтобы сообщить ядру, какую системную службу вы вызываете, поместите её номер в `EAX`. Разумеется, вам необходимо знать, что это за номер.

### 3.4.1. Файл syscalls

Номера перечислены в syscalls. Команда `locate syscalls` находит этот файл в нескольких различных форматах, все они создаются автоматически из syscalls.master.

Основной файл для стандартного соглашения о вызовах UNIX® можно найти в /usr/src/sys/kern/syscalls.master. Если вам необходимо использовать другое соглашение, реализованное в режиме эмуляции Linux, обратитесь к /usr/src/sys/i386/linux/syscalls.master.



Не только FreeBSD и Linux используют разные соглашения о вызовах, но иногда они используют разные номера для одних и тех же функций.

syscalls.master описывает, как должен быть выполнен вызов:

```
0   STD NOHIDE  { int nosys(void); } syscall nosys_args int
1   STD NOHIDE  { void exit(int rval); } exit rexit_args void
2   STD POSIX   { int fork(void); }
3   STD POSIX   { ssize_t read(int fd, void *buf, size_t nbyte); }
4   STD POSIX   { ssize_t write(int fd, const void *buf, size_t nbyte); }
5   STD POSIX   { int open(char *path, int flags, int mode); }
6   STD POSIX   { int close(int fd); }
etc...
```

Это крайний левый столбец, который указывает число, которое нужно поместить в `EAX`.

Самый правый столбец указывает, какие параметры нужно `втолкнуть` в стек командой push. Они `вталкиваются` *справа налево*.

Например, чтобы `открыть` файл, нам сначала нужно сделать `push` для `mode`, затем `flags`, а затем адрес, по которому хранится `path`.

# 4. Возвращаемые значения

От системных вызовов не было бы никакой пользы, если бы они не возвращали какое-либо значение: дескриптор открытого файла, количество байтов, прочитанных в буфер, системное время и т.д.

Кроме того, система должна уведомлять нас, если возникает ошибка: файл не существует, системные ресурсы исчерпаны, передан недопустимый параметр и т. д.

## 4.1. Страницы Справочника

Традиционным источником информации о различных системных вызовах в UNIX®-системах являются страницы Справочника. В FreeBSD системные вызовы описаны в разделе 2, иногда в разделе 3.

Например, open(2) говорит:

В случае успеха `open()` возвращает неотрицательное целое число, называемое файловым дескриптором. В случае ошибки возвращается `-1`, а переменной `errno` присваивается код ошибки.

Программист на ассемблере, впервые столкнувшийся с UNIX® и FreeBSD, сразу же задастся вопросом: где находится `errno` и как к ней обратиться?



Информация, представленная в руководствах, применима к программам на языке C. Программистам на языке ассемблера требуется дополнительная информация.

## 4.2. Где возвращаемые значения?

К сожалению, это зависит от ситуации… Для большинства системных вызовов возвращаемое значение находится в `EAX`, но не для всех. Хорошее правило при первой работе с системным вызовом — искать возвращаемое значение в `EAX`. Если его там нет, потребуется дополнительное исследование.



Я знаю о одном системном вызове, который возвращает значение в `EDX`: `SYS_fork`. Все остальные, с которыми я работал, используют `EAX`. Но я ещё не работал со всеми из них.



Если вы не можете найти ответ здесь или где-либо ещё, изучите исходный код libc и посмотрите, как он взаимодействует с ядром.

## 4.3. Где находится `errno`?

Фактически, нигде…

`errno` является частью языка C, а не ядра UNIX®. При прямом доступе к сервисам ядра код ошибки возвращается в регистре `EAX` — том же регистре, в котором обычно оказывается корректное возвращаемое значение.

Это совершенно логично. Если нет ошибки, то нет и кода ошибки. Если есть ошибка, то нет возвращаемого значения. Один регистр может содержать либо то, либо другое.

## 4.4. Определение возникновения ошибки

При использовании стандартного соглашения о вызовах FreeBSD флаг `carry flag` сбрасывается при успехе и устанавливается при неудаче.

При использовании режима эмуляции Linux знаковое значение в `EAX` неотрицательно в случае успеха и содержит возвращаемое значение. В случае ошибки значение отрицательное, т.е. `-errno`.

# 5. Создание переносимого кода

Портативность обычно не является сильной стороной языка ассемблера. Тем не менее, написание программ на ассемблере для разных платформ возможно, особенно с использованием nasm. Я создавал библиотеки на ассемблере, которые можно было собрать для таких разных операционных систем, как Windows® и FreeBSD.

Это становится ещё более возможным, когда вы хотите, чтобы ваш код работал на двух платформах, которые, хотя и различны, основаны на схожих архитектурах.

Например, FreeBSD — это UNIX®, а Linux — UNIX®-подобная система. Я упомянул лишь три различия между ними (с точки зрения программиста на ассемблере): соглашение о вызовах, номера функций и способ возврата значений.

## 5.1. Работа с номерами функций

Во многих случаях номера функций совпадают. Однако, даже если это не так, проблему легко решить: вместо использования чисел в коде применяйте константы, объявленные по-разному в зависимости от целевой архитектуры:

```
%ifdef  LINUX
%define SYS_execve  11
%else
%define SYS_execve  59
%endif
```

## 5.2. Работа с соглашениями

Оба, соглашение о вызовах и возвращаемое значение (проблема `errno`) могут быть решены с помощью макросов:

```
%ifdef  LINUX

%macro  system  0
    call    kernel
%endmacro

align 4
kernel:
    push    ebx
    push    ecx
    push    edx
    push    esi
    push    edi
    push    ebp

    mov ebx, [esp+32]
    mov ecx, [esp+36]
    mov edx, [esp+40]
    mov esi, [esp+44]
    mov ebp, [esp+48]
    int 80h

    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx

    or  eax, eax
    js  .errno
    clc
    ret

.errno:
    neg eax
    stc
    ret

%else

%macro  system  0
    int 80h
%endmacro
```

```
%endif
```

## 5.3. Устранение прочих проблем с переносимостью

Приведённые выше решения могут помочь в большинстве случаев написания кода, переносимого между FreeBSD и Linux. Тем не менее, с некоторыми сервисами ядра различия более глубокие.

В таком случае необходимо написать два разных обработчика для этих конкретных системных вызовов и использовать условную компиляцию. К счастью, большая часть вашего кода выполняет действия, отличные от вызовов ядра, поэтому обычно потребуется лишь несколько таких условных секций в коде.

## 5.4. Использование библиотеки

Вы можете полностью избежать проблем с переносимостью в основном коде, написав библиотеку системных вызовов. Создайте отдельную библиотеку для FreeBSD, другую для Linux и ещё другие библиотеки для дополнительных операционных систем.

В вашей библиотеке напишите отдельную функцию (или процедуру, если вы предпочитаете традиционную терминологию ассемблера) для каждого системного вызова. Используйте соглашение о вызовах C для передачи параметров. Однако по-прежнему передавайте номер вызова через `EAX`. В таком случае ваша библиотека FreeBSD может быть очень простой, так как множество внешне различных функций могут быть просто метками одного и того же кода:

```
sys.open:
sys.close:
[etc...]
    int 80h
    ret
```

Ваша библиотека Linux потребует больше различных функций. Но даже здесь вы можете группировать системные вызовы, используя одинаковое количество параметров:

```
sys.exit:
sys.close:
[etc... one-parameter functions]
    push    ebx
    mov ebx, [esp+12]
    int 80h
    pop ebx
    jmp sys.return

...

sys.return:
```

```
    or  eax, eax
    js  sys.err
    clc
    ret

sys.err:
    neg eax
    stc
    ret
```

Подход с использованием библиотек может показаться неудобным на первый взгляд, так как требует создания отдельного файла, от которого зависит ваш код. Однако у него есть множество преимуществ: во-первых, вам нужно написать его лишь один раз, и затем вы можете использовать его во всех своих программах. Вы даже можете позволить другим программистам на ассемблере использовать его или, возможно, воспользоваться библиотекой, написанной кем-то другим. Но, пожалуй, самое большое преимущество библиотеки заключается в том, что ваш код может быть перенесён на другие системы, даже другими программистами, просто путём написания новой библиотеки без каких-либо изменений в вашем коде.

Если вам не нравится идея использования библиотеки, вы можете хотя бы разместить все системные вызовы в отдельном файле на ассемблере и скомпоновать его с основной программой. Здесь, опять же, все, что нужно сделать переносчикам, — это создать новый объектный файл для компоновки с основной программой.

## 5.5. Использование включаемого файла

Если вы выпускаете своё программное обеспечение в виде исходного кода (или вместе с ним), вы можете использовать макросы и размещать их в отдельном файле, который включается в ваш код.

Портеры вашего программного обеспечения просто напишут новый include-файл. Никакая библиотека или внешний объектный файл не требуются, и ваш код остаётся переносимым без необходимости редактирования.



Это подход, который мы будем использовать на протяжении всей главы. Мы назовем наш включаемый файл system.inc и будем добавлять в него новые системные вызовы по мере их рассмотрения.

Мы можем начать наш system.inc с объявления стандартных файловых дескрипторов:

```
%define stdin   0
%define stdout  1
%define stderr  2
```

Далее мы создаем символическое имя для каждого системного вызова:

```
%define SYS_nosys   0
%define SYS_exit    1
%define SYS_fork    2
%define SYS_read    3
%define SYS_write   4
; [etc...]
```

Добавляем короткую, неглобальную процедуру с длинным именем, чтобы случайно не использовать это имя в нашем коде:

```
section .text
align 4
access.the.bsd.kernel:
    int 80h
    ret
```

Мы создаем макрос, который принимает один аргумент — номер системного вызова:

```
%macro  system  1
    mov eax, %1
    call    access.the.bsd.kernel
%endmacro
```

Наконец, мы создаем макросы для каждого системного вызова. Эти макросы не принимают аргументов.

```
%macro  sys.exit    0
    system  SYS_exit
%endmacro

%macro  sys.fork    0
    system  SYS_fork
%endmacro

%macro  sys.read    0
    system  SYS_read
%endmacro

%macro  sys.write   0
    system  SYS_write
%endmacro

; [etc...]
```

Продолжайте, введите это в ваш редактор и сохраните как system.inc. Мы добавим больше по мере обсуждения дополнительных системных вызовов.

# 6. Наша первая программа

Мы готовы к нашей первой обязательной программе — Hello, World!

```
    %include    'system.inc'

    section .data
    hello   db  'Hello, World!', 0Ah
    hbytes  equ $-hello

    section .text
    global  _start
_start:
    push    dword hbytes
    push    dword hello
    push    dword stdout
    sys.write

    push    dword 0
    sys.exit
```

Вот что он делает: Строка 1 включает определения, макросы и код из файла system.inc.

Строки 3-5 содержат данные: строка 3 начинает раздел/сегмент данных. Строка 4 содержит строку "Hello, World!", за которой следует новая строка (`0Ah`). Строка 5 создаёт константу, содержащую длину строки из строки 4 в байтах.

Строки 7-16 содержат код. Обратите внимание, что FreeBSD использует формат файлов *elf* для исполняемых файлов, который требует, чтобы каждая программа запускается с адреса, помеченного как `_start` (или, точнее, компоновщик ожидает этого). Эта метка должна быть глобальной.

Строки 10-13 указывают системе записать `hbytes` байтов строки `hello` в `stdout`.

Строки 15-16 указывают системе завершить программу с возвращаемым значением `0`. Системный вызов `SYS_exit` никогда не возвращает управление, поэтому код завершается в этой точке.



Если вы перешли на UNIX® с опытом программирования на ассемблере для MS-DOS®, вы, возможно, привыкли писать напрямую в видеопамять. В FreeBSD или любой другой разновидности UNIX® вам не придётся об этом беспокоиться. С вашей точки зрения, вы записываете данные в файл под названием stdout. Это может быть экран, терминал telnet, обычный файл или даже входные данные другой программы. Определять, что именно это будет, — задача системы.

## 6.1. Ассемблирование кода

Наберите код в редакторе и сохраните его в файле с именем hello.asm. Для сборки вам понадобится nasm.

### 6.1.1. Установка nasm

Если у вас нет nasm, введите:

```
% su
Password:your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

Вы можете ввести `make install clean` вместо просто `make install`, если не хотите сохранять исходный код nasm.

В любом случае FreeBSD автоматически загрузит nasm из интернета, скомпилирует его и установит в вашу систему.



Если ваша система не FreeBSD, вам нужно получить nasm с его домашней страницы. Вы по-прежнему можете использовать его для ассемблирования кода FreeBSD.

Теперь вы можете собрать, скомпоновать и запустить код:

```
% nasm -f elf hello.asm
% ld -s -o hello hello.o
% ./hello
Hello, World!
%
```

# 7. Написание фильтров UNIX®

Распространённым типом приложений в UNIX® являются фильтры — программы, которые читают данные из stdin, обрабатывают их определённым образом, а затем записывают результат в stdout.

В этой главе мы разработаем простой фильтр и научимся читать из stdin и писать в stdout. Этот фильтр будет преобразовывать каждый байт входных данных в шестнадцатеричное число, за которым следует пробел.

```
%include    'system.inc'
```

```
section .data
hex db  '0123456789ABCDEF'
buffer  db  0, 0, ' '

section .text
global  _start
_start:
    ; read a byte from stdin
    push    dword 1
    push    dword buffer
    push    dword stdin
    sys.read
    add esp, byte 12
    or  eax, eax
    je  .done

    ; convert it to hex
    movzx   eax, byte [buffer]
    mov edx, eax
    shr dl, 4
    mov dl, [hex+edx]
    mov [buffer], dl
    and al, 0Fh
    mov al, [hex+eax]
    mov [buffer+1], al

    ; print it
    push    dword 3
    push    dword buffer
    push    dword stdout
    sys.write
    add esp, byte 12
    jmp short _start

.done:
    push    dword 0
    sys.exit
```

В разделе данных мы создаем массив с именем `hex`. Он содержит 16 шестнадцатеричных цифр в порядке возрастания. За массивом следует буфер, который мы будем использовать как для ввода, так и для вывода. Первые два байта буфера изначально установлены в `0`. Именно сюда мы будем записывать две шестнадцатеричные цифры (первый байт также является местом, откуда мы будем считывать ввод). Третий байт — это пробел.

Фрагмент кода состоит из четырёх частей: чтение байта, преобразование его в шестнадцатеричное число, запись результата и завершение программы.

Для чтения байта мы просим систему прочитать один байт из stdin и сохранить его в первом байте `buffer`. Система возвращает количество прочитанных байтов в `EAX`. Это значение будет `1`, пока поступают данные, или `0`, если больше нет доступных входных

данных. Поэтому мы проверяем значение EAX. Если оно равно 0, мы переходим к метке .done, в противном случае продолжаем выполнение.



Для простоты мы пока игнорируем возможность возникновения ошибки.

Шестнадцатеричное преобразование считывает байт из buffer в EAX, а точнее только в AL, обнуляя остальные биты EAX. Мы также копируем байт в EDX, потому что нам нужно преобразовать верхние четыре бита (ниббл) отдельно от нижних четырёх битов. Результат сохраняется в первых двух байтах буфера.

Далее мы просим систему записать три байта буфера, то есть две шестнадцатеричные цифры и пробел, в stdout. Затем мы возвращаемся к началу программы и обрабатываем следующий байт.

Когда ввод больше не остаётся, мы просим систему завершить нашу программу, возвращая ноль, что традиционно означает успешное выполнение программы.

Продолжайте и сохраните код в файле с именем hex.asm, затем введите следующее (символ ^D означает, что нужно нажать клавишу управления и, удерживая её, ввести D):

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A ^D %
```



Если вы переходите на UNIX® с MS-DOS®, вам может быть интересно, почему каждая строка заканчивается на 0A вместо 0D 0A. Это связано с тем, что UNIX® не использует соглашение cr/lf, а использует соглашение "новая строка", которое в шестнадцатеричном виде представлено как 0A.

Можем ли мы это улучшить? Что ж, во-первых, это немного запутанно, потому что после преобразования строки текста наш ввод больше не начинается с начала строки. Мы можем изменить это, чтобы после каждого 0A выводилась новая строка вместо пробела:

```
%include    'system.inc'

section .data
hex db  '0123456789ABCDEF'
buffer  db  0, 0, ' '

section .text
global  _start
_start:
    mov cl, ' '

.loop:
```

```
    ; read a byte from stdin
    push    dword 1
    push    dword buffer
    push    dword stdin
    sys.read
    add esp, byte 12
    or  eax, eax
    je  .done

    ; convert it to hex
    movzx   eax, byte [buffer]
    mov [buffer+2], cl
    cmp al, 0Ah
    jne .hex
    mov [buffer+2], al

.hex:
    mov edx, eax
    shr dl, 4
    mov dl, [hex+edx]
    mov [buffer], dl
    and al, 0Fh
    mov al, [hex+eax]
    mov [buffer+1], al

    ; print it
    push    dword 3
    push    dword buffer
    push    dword stdout
    sys.write
    add esp, byte 12
    jmp short .loop

.done:
    push    dword 0
    sys.exit
```

Мы сохранили пробел в регистре `CL`. Это безопасно, потому что, в отличие от Microsoft® Windows®, вызовы системы UNIX® не изменяют значение регистров, которые не используются для возврата значения.

Это означает, что нам нужно установить `CL` только один раз. Поэтому мы добавили новую метку `.loop` и переходим к ней для следующего байта вместо перехода к `_start`. Мы также добавили метку `.hex`, чтобы третий байт `buffer` мог быть либо пробелом, либо новой строкой.

После внесения изменений в файл hex.asm введите:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
```

```
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Выглядит лучше. Но этот код довольно неэффективен! Мы выполняем системный вызов для каждого отдельного байта дважды (один раз для чтения и ещё один для записи вывода).

# 8. Буферизованный ввод и вывод

Мы можем повысить эффективность нашего кода, буферизуя ввод и вывод. Мы создаём входной буфер и читаем сразу целую последовательность байтов. Затем мы извлекаем их по одному из буфера.

Мы также создаем выходной буфер. Мы сохраняем наш вывод в нём, пока он не заполнится. В этот момент мы просим ядро записать содержимое буфера в stdout.

Программа завершается, когда больше нет входных данных. Но нам всё ещё нужно попросить ядро записать содержимое нашего выходного буфера в stdout в последний раз, иначе часть нашего вывода попадёт в буфер, но так и не будет отправлена. Не забудьте об этом, иначе будете недоумевать, куда пропала часть вывода.

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
hex db  '0123456789ABCDEF'

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
global  _start
_start:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from stdin
    call    getchar

    ; convert it to hex
    mov dl, al
```

```
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword stdin
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer
    push    dword 0
    sys.exit

align 4
putchar:
```

```
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword stdout
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

Теперь у нас есть третий раздел в исходном коде с именем `.bss`. Этот раздел не включается в исполняемый файл и, следовательно, не может быть инициализирован. Мы используем `resb` вместо `db`. Это просто резервирует запрошенный размер неинициализированной памяти для нашего использования.

Мы используем тот факт, что система не изменяет регистры: мы используем регистры для того, что в противном случае пришлось бы хранить в глобальных переменных в секции `.data`. Именно поэтому соглашение UNIX® о передаче параметров системных вызовов через стек превосходит соглашение Microsoft о передаче их в регистрах: мы можем оставить регистры для собственного использования.

Мы используем `EDI` и `ESI` как указатели на следующий байт для чтения или записи. Мы используем `EBX` и `ECX` для отслеживания количества байтов в двух буферах, чтобы знать, когда нужно вывести данные в систему или считать новые данные из системы.

Давайте посмотрим, как это работает сейчас:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
Here I come!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Не то, что вы ожидали? Программа не выводила результат, пока мы не нажали `^D`. Это легко исправить, добавив три строки кода для вывода результата каждый раз, когда мы преобразуем новую строку в `0A`. Я пометил эти три строки символом > (не копируйте > в ваш hex.asm).

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
hex db  '0123456789ABCDEF'

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
global  _start
_start:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from stdin
    call    getchar

    ; convert it to hex
    mov dl, al
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
>    cmp al, 0Ah
>    jne .loop
>    call    write
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
    jne .fetch
```

```
    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword stdin
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword stdout
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

Теперь давайте посмотрим, как это работает:

```
% nasm -f elf hex.asm
```

```
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Неплохо для исполняемого файла размером 644 байта, не так ли!



Такой подход к буферизированному вводу/выводу всё ещё содержит скрытую опасность. Я расскажу об этом и исправлю её позже, когда речь пойдёт о тёмной стороне буферизации.

# 8.1. Как отменить чтение символа



Это может быть несколько сложной темой, в основном представляющей интерес для программистов, знакомых с теорией компиляторов. Если хотите, вы можете перейти к следующему разделу, и, возможно, прочитаете это позже.

Хотя наш пример программы не требует этого, более сложные фильтры часто нуждаются в предварительном просмотре. Другими словами, им может потребоваться узнать, какой следующий символ (или даже несколько символов). Если следующий символ имеет определённое значение, он является частью текущего обрабатываемого токена. В противном случае — нет.

Например, вы можете анализировать входной поток на наличие текстовой строки (например, при реализации компилятора языка): если символ следует за другим символом или, возможно, цифрой, он является частью обрабатываемой лексемы. Если за ним следует пробел или другое значение, то он не является частью текущей лексемы.

Это представляет интересную проблему: как вернуть следующий символ обратно во входной поток, чтобы его можно было прочитать позже?

Одно из возможных решений — сохранить его в символьной переменной, а затем установить флаг. Мы можем изменить `getchar`, чтобы он проверял флаг, и если он установлен, извлекал байт из этой переменной вместо буфера ввода, а затем сбрасывал флаг. Но, конечно, это замедляет работу.

В языке C есть функция `ungetc()`, как раз для этой цели. Есть ли быстрый способ реализовать её в нашем коде? Я хочу, чтобы вы пролистали назад и взглянули на процедуру `getchar`, и попробовали найти красивое и быстрое решение, прежде чем читать следующий абзац. Затем вернитесь сюда и посмотрите моё собственное решение.

Ключом к возвращению символа обратно в поток является то, как мы получаем символы изначально:

Сначала проверяем, пуст ли буфер, проверяя значение EBX. Если оно равно нулю, вызываем процедуру read.

Если у нас есть доступный символ, мы используем lodsb, затем уменьшаем значение EBX. Инструкция lodsb фактически идентична:

```
mov al, [esi]
    inc esi
```

Байт, который мы извлекли, остаётся в буфере до следующего вызова read. Мы не знаем, когда это произойдет, но знаем, что этого не случится до следующего вызова getchar. Следовательно, чтобы "вернуть" последний прочитанный байт обратно в поток, нам достаточно уменьшить значение ESI и увеличить значение EBX:

```
ungetc:
    dec esi
    inc ebx
    ret
```

Но будьте осторожны! Мы в полной безопасности, если заглядываем вперёд только на один символ за раз. Если же мы проверяем несколько следующих символов и вызываем ungetc несколько раз подряд, это будет работать в большинстве случаев, но не всегда (и ошибки будет сложно отладить). Почему?

Потому что пока getchar не вызывает read, все предварительно прочитанные байты остаются в буфере, и наш ungetc работает без сбоев. Но как только getchar вызывает read, содержимое буфера изменяется.

Мы всегда можем рассчитывать на корректную работу ungetc с последним символом, прочитанным через getchar, но не с любым символом, прочитанным до этого.

Если ваша программа читает более одного байта вперёд, у вас есть как минимум два варианта:

Если возможно, измените программу так, чтобы она читала только один байт вперёд. Это самое простое решение.

Если эта опция недоступна, сначала определите максимальное количество символов, которое вашей программе может потребоваться вернуть во входной поток за один раз. Увеличьте это число немного, чтобы быть уверенным, предпочтительно до кратного 16 — так оно будет лучше выровнено. Затем измените секцию .bss в вашем коде и создайте небольшой "запасной" буфер прямо перед вашим входным буфером, примерно так:

```
section .bss
    resb    16  ; or whatever the value you came up with
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE
```

Вам также необходимо изменить ваш `ungetc`, чтобы передать значение байта для возврата в `AL`:

```
ungetc:
    dec esi
    inc ebx
    mov [esi], al
    ret
```

С этим изменением вы можете безопасно вызывать `ungetc` до 17 раз подряд (первый вызов всё ещё будет в пределах буфера, остальные 16 могут быть либо в пределах буфера, либо в пределах "запасного" пространства).

# 9. Аргументы командной строки

Наша программа hex будет полезнее, если она сможет читать имена входного и выходного файлов из командной строки, т.е. если она сможет обрабатывать аргументы командной строки. Но… Где они?

Прежде чем UNIX® система запустит программу, она делает `push` для некоторых данных, помещая их в стек, затем переходит к метке `_start` программы. Да, я сказал "переходит", а не "вызывает". Это означает, что данные можно прочитать с помощью `[esp+offset]` или просто сделать `pop` для них.

Значение на вершине стека содержит количество аргументов командной строки. Оно традиционно называется `argc`, что означает "argument count".

Далее следуют аргументы командной строки, все `argc` штук. Обычно их называют `argv`, что означает "значение(я) аргумента". То есть мы получаем `argv[0]`, `argv[1]`, ⋯, `argv[argc-1]`. Это не сами аргументы, а указатели на аргументы, то есть адреса памяти, где находятся реальные аргументы. Сами аргументы представляют собой строки символов, завершающиеся нулевым символом ('\0').

Список `argv` завершается указателем NULL, который представляет собой просто `0`. Есть и другие детали, но пока этого достаточно для наших целей.



Если вы перешли из среды программирования MS-DOS®, основное различие заключается в том, что каждый аргумент находится в отдельной строке. Второе различие состоит в том, что нет практического ограничения на количество аргументов.

Вооружившись этими знаниями, мы почти готовы к следующей версии hex.asm. Однако сначала нам нужно добавить несколько строк в system.inc:

Сначала нам нужно добавить две новые записи в наш список номеров системных вызовов:

```
%define SYS_open    5
```

```
%define SYS_close   6
```

Затем мы добавляем два новых макроса в конце файла:

```
%macro  sys.open    0
    system  SYS_open
%endmacro

%macro  sys.close   0
    system  SYS_close
%endmacro
```

Вот наш измененный исходный код:

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
fd.in   dd  stdin
fd.out  dd  stdout
hex db  '0123456789ABCDEF'

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
align 4
err:
    push    dword 1     ; return failure
    sys.exit

align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]

    pop ecx
    jecxz   .init       ; no more arguments

    ; ECX contains the path to input file
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  err     ; open failed

    add esp, byte 8
    mov [fd.in], eax
```

```
    pop ecx
    jecxz   .init       ; no more arguments

    ; ECX contains the path to output file
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
    jc  err

    add esp, byte 12
    mov [fd.out], eax

.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from input file or stdin
    call    getchar

    ; convert it to hex
    mov dl, al
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
    cmp al, dl
    jne .loop
    call    write
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
```

```
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close

    ; return success
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
```

```
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

В нашем разделе `.data` теперь есть две новые переменные, `fd.in` и `fd.out`. Здесь мы сохраняем дескрипторы файлов для ввода и вывода.

В разделе `.text` мы заменили ссылки с `stdin` и `stdout` на `[fd.in]` и `[fd.out]`.

Раздел `.text` теперь начинается с простого обработчика ошибок, который просто завершает программу с кодом возврата `1`. Обработчик ошибок расположен перед `_start`, чтобы находиться вблизи от места возникновения ошибок.

Естественно, выполнение программы по-прежнему начинается с `_start`. Сначала мы удаляем `argc` и `argv[0]` из стека: они не представляют для нас интереса (по крайней мере, в этой программе).

Мы помещаем `argv[1]` в `ECX`. Этот регистр особенно подходит для указателей, так как мы можем обрабатывать NULL-указатели с помощью `jecxz`. Если `argv[1]` не равен NULL, мы пытаемся открыть файл с именем, указанным в первом аргументе. В противном случае продолжаем программу как раньше: чтение из `stdin`, запись в `stdout`. Если нам не удаётся открыть входной файл (например, он не существует), мы переходим к обработчику ошибок и завершаем работу.

Если всё прошло успешно, мы проверяем второй аргумент. Если он присутствует, мы открываем выходной файл. В противном случае, мы отправляем вывод в `stdout`. Если нам не удаётся открыть выходной файл (например, он существует и у нас нет прав на запись), мы снова переходим к обработчику ошибок.

Остальная часть кода остаётся прежней, за исключением того, что мы закрываем входной и выходной файлы перед завершением, и, как упоминалось, используем `[fd.in]` и `[fd.out]`.

Наш исполняемый файл теперь имеет внушительный размер в 768 байт.

Можем ли мы улучшить его ещё? Конечно! Каждую программу можно улучшить. Вот несколько идей, что мы могли бы сделать:

- Сделать наш обработчик ошибок, выводящий сообщение в `stderr`.
- Добавить обработчики ошибок в функции `read` и `write`.
- Закрывать `stdin` при открытии входного файла, `stdout` при открытии выходного файла.
- Добавить параметры командной строки, такие как `-i` и `-o`, чтобы можно было перечислять входные и выходные файлы в любом порядке или, возможно, читать из `stdin` и записывать в файл.
- Выводить сообщение с подсказкой об использовании программы, если аргументы командной строки указаны неверно.

Я оставлю эти улучшения в качестве упражнения для читателя: вы уже знаете всё необходимое для их реализации.

# 10. Окружение UNIX®

Важным концептом UNIX® является окружение, которое определяется *переменными окружения*. Некоторые из них устанавливаются системой, другие — пользователем, третьи — оболочкой или любой программой, которая загружает другую программу.

## 10.1. Как найти переменные окружения

Я говорил ранее, что когда программа начинает выполняться, в стеке находятся `argc`, за которым следует массив `argv`, завершающийся NULL, а затем что-то ещё. Это "что-то ещё" — это *окружение*, или, если быть точнее, массив указателей на *переменные окружения*, завершающийся NULL. Это часто называют `env`.

Структура `env` такая же, как у `argv` — список адресов памяти, заканчивающийся NULL (`0`). В данном случае нет `"envc"` — конец массива определяется поиском последнего NULL.

Переменные обычно имеют формат `name=value`, но иногда часть `=value` может отсутствовать. Необходимо учитывать эту вероятность.

## 10.2. webvars

Я мог бы просто показать вам код, который выводит окружение так же, как команда UNIX® env. Но я подумал, что будет интереснее написать простую CGI-утилиту на ассемблере.

### 10.2.1. CGI: краткий обзор

У меня есть подробное руководство по CGI на моем веб-сайте, но вот очень краткий обзор CGI:

- Веб-сервер взаимодействует с CGI-программой, устанавливая *переменные окружения*.
- Программа CGI отправляет свой вывод в stdout. Веб-сервер считывает его оттуда.
- Он должен начинаться с HTTP-заголовка, за которым следуют две пустые строки.
- Затем он выводит HTML-код или любые другие данные, которые он генерирует.



В то время как некоторые *переменные окружения* используют стандартные имена, другие различаются в зависимости от веб-сервера. Это делает программу webvars весьма полезным инструментом для диагностики.

### 10.2.2. Код

Наша программа webvars, таким образом, должна отправить HTTP-заголовок, за которым следует HTML-разметка. Затем она должна прочитать *переменные окружения* одну за другой и отправить их как часть HTML-страницы.

Код приведен ниже. Я разместил комментарии и пояснения прямо в коде:

```
;;;;;;; webvars.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Copyright (c) 2000 G. Adam Stanislav
; All rights reserved.
;
; Redistribution and use in source and binary forms, with or without
; modification, are permitted provided that the following conditions
; are met:
; 1. Redistributions of source code must retain the above copyright
;    notice, this list of conditions and the following disclaimer.
; 2. Redistributions in binary form must reproduce the above copyright
;    notice, this list of conditions and the following disclaimer in the
;    documentation and/or other materials provided with the distribution.
;
; THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
; ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
; ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
; FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
; DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
; OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
; HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
; LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
; OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
; SUCH DAMAGE.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Version 1.0
;
; Started:   8-Dec-2000
; Updated:   8-Dec-2000
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include    'system.inc'

section .data
http    db  'Content-type: text/html', 0Ah, 0Ah
    db  '<?xml version="1.0" encoding="utf-8"?>', 0Ah
    db  '<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML Strict//EN" '
    db  '"DTD/xhtml1-strict.dtd">', 0Ah
    db  '<html xmlns="http://www.w3.org/1999/xhtml" '
    db  'xml.lang="en" lang="en">', 0Ah
    db  '<head>', 0Ah
    db  '<title>Web Environment</title>', 0Ah
    db  '<meta name="author" content="G. Adam Stanislav" />', 0Ah
    db  '</head>', 0Ah, 0Ah
    db  '<body bgcolor="#ffffff" text="#000000" link="#0000ff" '
    db  'vlink="#840084" alink="#0000ff">', 0Ah
```

```
    db  '<div class="webvars">', 0Ah
    db  '<h1>Web Environment</h1>', 0Ah
    db  '<p>The following <b>environment variables</b> are defined '
    db  'on this web server:</p>', 0Ah, 0Ah
    db  '<table align="center" width="80" border="0" cellpadding="10" '
    db  'cellspacing="0" class="webvars">', 0Ah
httplen equ $-http
left    db  '<tr>', 0Ah
    db  '<td class="name"><tt>'
leftlen equ $-left
middle  db  '</tt></td>', 0Ah
    db  '<td class="value"><tt><b>'
midlen  equ $-middle
undef   db  '<i>(undefined)</i>'
undeflen    equ $-undef
right   db  '</b></tt></td>', 0Ah
    db  '</tr>', 0Ah
rightlen    equ $-right
wrap    db  '</table>', 0Ah
    db  '</div>', 0Ah
    db  '</body>', 0Ah
    db  '</html>', 0Ah, 0Ah
wraplen equ $-wrap

section .text
global  _start
_start:
    ; First, send out all the http and xhtml stuff that is
    ; needed before we start showing the environment
    push    dword httplen
    push    dword http
    push    dword stdout
    sys.write

    ; Now find how far on the stack the environment pointers
    ; are. We have 12 bytes we have pushed before "argc"
    mov eax, [esp+12]

    ; We need to remove the following from the stack:
    ;
    ;   The 12 bytes we pushed for sys.write
    ;   The  4 bytes of argc
    ;   The EAX*4 bytes of argv
    ;   The  4 bytes of the NULL after argv
    ;
    ; Total:
    ;   20 + eax * 4
    ;
    ; Because stack grows down, we need to ADD that many bytes
    ; to ESP.
    lea esp, [esp+20+eax*4]
```

```
    cld     ; This should already be the case, but let's be sure.

    ; Loop through the environment, printing it out
.loop:
    pop edi
    or  edi, edi    ; Done yet?
    je  near .wrap

    ; Print the left part of HTML
    push    dword leftlen
    push    dword left
    push    dword stdout
    sys.write

    ; It may be tempting to search for the '=' in the env string next.
    ; But it is possible there is no '=', so we search for the
    ; terminating NUL first.
    mov esi, edi    ; Save start of string
    sub ecx, ecx
    not ecx     ; ECX = FFFFFFFF
    sub eax, eax
repne   scasb
    not ecx     ; ECX = string length + 1
    mov ebx, ecx    ; Save it in EBX

    ; Now is the time to find '='
    mov edi, esi    ; Start of string
    mov al, '='
repne   scasb
    not ecx
    add ecx, ebx    ; Length of name

    push    ecx
    push    esi
    push    dword stdout
    sys.write

    ; Print the middle part of HTML table code
    push    dword midlen
    push    dword middle
    push    dword stdout
    sys.write

    ; Find the length of the value
    not ecx
    lea ebx, [ebx+ecx-1]

    ; Print "undefined" if 0
    or  ebx, ebx
    jne .value
```

```
    mov ebx, undeflen
    mov edi, undef

.value:
    push    ebx
    push    edi
    push    dword stdout
    sys.write

    ; Print the right part of the table row
    push    dword rightlen
    push    dword right
    push    dword stdout
    sys.write

    ; Get rid of the 60 bytes we have pushed
    add esp, byte 60

    ; Get the next variable
    jmp .loop

.wrap:
    ; Print the rest of HTML
    push    dword wraplen
    push    dword wrap
    push    dword stdout
    sys.write

    ; Return success
    push    dword 0
    sys.exit
```

Этот код создаёт исполняемый файл размером 1 396 байт. Большая его часть — это данные, а именно HTML-разметка, которую нам нужно отправить.

Запустите ассемблер и слинкуйте как обычно:

```
% nasm -f elf webvars.asm
% ld -s -o webvars webvars.o
```

Для использования необходимо загрузить webvars на ваш веб-сервер. В зависимости от настроек веб-сервера, возможно, потребуется разместить его в специальном каталоге cgi-bin или переименовать с расширением .cgi.

Затем вам нужно использовать браузер для просмотра вывода. Чтобы увидеть вывод на моем веб-сервере, перейдите по ссылке http://www.int80h.org/webvars/. Если вам интересно узнать о дополнительных переменных окружения в защищенном паролем веб-каталоге, перейдите по адресу http://www.int80h.org/private/, используя имя `asm` и пароль `programmer`.

# 11. Работа с файлами

Мы уже выполнили некоторые базовые операции с файлами: мы знаем, как их открывать и закрывать, как читать и записывать их с использованием буферов. Однако UNIX® предлагает гораздо больше возможностей при работе с файлами. В этом разделе мы рассмотрим некоторые из них и в итоге создадим удобную утилиту для преобразования файлов.

В самом деле, начнем с конца, то есть с утилиты преобразования файлов. Всегда легче программировать, когда с самого начала известно, каким должен быть конечный продукт.

Одной из первых программ, которые я написал для UNIX®, была tuc — конвертер текста в файл UNIX®. Она преобразует текстовый файл из других операционных систем в текстовый файл UNIX®. Другими словами, она изменяет различные виды окончаний строк на стандартные для UNIX®. Результат сохраняется в другом файле. По желанию, она может преобразовать текстовый файл UNIX® в текстовый файл DOS.

Я широко использовал `tuc`, но всегда только для преобразования из какой-либо другой ОС в UNIX®, никогда наоборот. Мне всегда хотелось, чтобы он просто перезаписывал файл, вместо того чтобы мне приходилось отправлять вывод в другой файл. В большинстве случаев я в итоге использую его так:

```
% tuc myfile tempfile
% mv tempfile myfile
```

Было бы здорово иметь ftuc, т.е., *быстрый tuc*, и использовать его вот так:

```
% ftuc myfile
```

В этой главе мы напишем ftuc на языке ассемблера (оригинальный tuc написан на C) и в процессе изучим различные файловые сервисы ядра.

На первый взгляд, такое преобразование файла кажется очень простым: нужно всего лишь удалить символы возврата каретки, верно?

Если вы ответили «да», подумайте ещё раз: такой подход будет работать в большинстве случаев (по крайней мере, с текстовыми файлами MS DOS), но иногда он будет давать сбой.

Проблема в том, что не все текстовые файлы, не относящиеся к UNIX®, завершают строки последовательностью возврата каретки / перевода строки. Некоторые используют возврат каретки без перевода строки. Другие объединяют несколько пустых строк в один возврат каретки, за которым следует несколько переводов строки. И так далее.

Конвертер текстовых файлов, следовательно, должен уметь обрабатывать любые возможные окончания строк:

- возврат каретки (carriage return) / перевод строки (line feed)

- возврат каретки
- перевод строки / возврат каретки
- перевод строки

Это также должно обрабатывать файлы, использующие комбинации вышеуказанного (например, возврат каретки с последующими несколькими переводами строки).

## 11.1. Конечный автомат

Проблема легко решается с использованием техники, называемой *конечный автомат*, изначально разработанной создателями цифровых электронных схем. *Конечный автомат* — это цифровая схема, выход которой зависит не только от входа, но и от предыдущего входа, то есть от её состояния. Микропроцессор является примером *конечного автомата*: наш код на языке ассемблера транслируется в машинный язык, где одни инструкции ассемблера превращаются в один байт машинного кода, а другие — в несколько байтов. Когда микропроцессор извлекает байты из памяти один за другим, некоторые из них просто изменяют его состояние, а не производят какой-либо выходной сигнал. После извлечения всех байтов кода операции микропроцессор выдаёт выходной сигнал, изменяет значение регистра и т. д.

Из-за этого всё программное обеспечение по сути представляет собой последовательность инструкций состояния для микропроцессора. Тем не менее, концепция *конечного автомата* также полезна при проектировании программного обеспечения.

Наш конвертер текстовых файлов можно представить в виде *конечного автомата* с тремя возможными состояниями. Мы могли бы назвать их состояниями 0-2, но будет проще, если дадим им символические имена:

- ordinary
- cr
- lf

Наша программа начнёт работу в обычном состоянии. В этом состоянии действие программы зависит от её входных данных следующим образом:

- Если ввод представляет собой что-либо, кроме возврата каретки или перевода строки, ввод просто передаётся на вывод. Состояние остаётся неизменным.
- Если входной символ — возврат каретки, состояние изменяется на cr. Затем входной символ отбрасывается, т.е. вывод не производится.
- Если входной символ является переводом строки, состояние изменяется на lf. Затем входной символ отбрасывается.

Всякий раз, когда мы находимся в состоянии `cr`, это означает, что последним вводом был символ возврата каретки, который не был обработан. Действия нашего программного обеспечения в этом состоянии снова зависят от текущего ввода:

- Если ввод отличается от возврата каретки или перевода строки, вывести перевод

строки, затем вывести ввод, а затем изменить состояние на обычное.

- Если входной символ — возврат каретки, значит, мы получили два (или более) возврата каретки подряд. Мы отбрасываем ввод, выводим перевод строки и оставляем состояние неизменным.
- Если входной символ — это перевод строки, мы выводим перевод строки и меняем состояние на обычное. Обратите внимание, что это не то же самое, что в первом случае выше — если бы мы попытались объединить их, мы бы выводили два перевода строки вместо одного.

Наконец, мы находимся в состоянии `lf` после получения перевода строки, которому не предшествовал возврат каретки. Это произойдет, если наш файл уже в формате UNIX®, или когда несколько строк подряд выражены одним возвратом каретки, за которым следуют несколько переводов строк, или когда строка заканчивается последовательностью перевода строки / возврата каретки. Вот как нам нужно обрабатывать ввод в этом состоянии:

- Если ввод отличается от возврата каретки или перевода строки, мы выводим перевод строки, затем выводим ввод и изменяем состояние на обычное. Это действие полностью совпадает с действием в состоянии `cr` при получении аналогичного ввода.
- Если ввод представляет собой символ возврата каретки, мы отбрасываем ввод, выводим символ перевода строки, затем изменяем состояние на обычное.
- Если входной символ — перевод строки, мы выводим перевод строки и оставляем состояние неизменным.

### 11.1.1. Конечное состояние

Приведённый выше *конечный автомат* работает для всего файла, но оставляет возможность, что последний конец строки будет проигнорирован. Это произойдёт, если файл заканчивается одиночным возвратом каретки или одиночным переводом строки. Я не подумал об этом, когда писал tuc, и лишь позже обнаружил, что иногда он удаляет последний конец строки.

Эта проблема легко решается проверкой состояния после обработки всего файла. Если состояние не является обычным, нам просто нужно вывести последний перевод строки.



Теперь, когда мы выразили наш алгоритм в виде *конечного автомата*, мы могли бы легко разработать специализированную цифровую электронную схему («чип») для выполнения преобразования. Конечно, это было бы значительно дороже, чем написание программы на языке ассемблера.

### 11.1.2. Счетчик вывода

Поскольку наша программа преобразования файлов может объединять два символа в один, нам необходимо использовать счётчик вывода. Мы инициализируем его значением `0` и увеличиваем каждый раз, когда отправляем символ на выход. В конце программы счётчик укажет, какой размер необходимо установить для файла.

# 11.2. Реализация конечного автомата в программном обеспечении

Самая сложная часть работы с *конечным автоматом* — это анализ задачи и её представление в виде *конечного автомата*. После этого программное обеспечение практически пишется само.

На языке высокого уровня, таком как C, существует несколько основных подходов. Один из них — использование оператора `switch`, который выбирает, какую функцию следует выполнить. Например,

```
switch (state) {
    default:
    case REGULAR:
        regular(inputchar);
        break;
    case CR:
        cr(inputchar);
        break;
    case LF:
        lf(inputchar);
        break;
    }
```

Еще один подход заключается в использовании массива указателей на функции, например:

```
(output[state])(inputchar);
```

Еще один вариант — сделать `state` указателем на функцию, установив его на соответствующую функцию:

```
(*state)(inputchar);
```

Это подход, который мы будем использовать в нашей программе, потому что его очень легко реализовать на языке ассемблера, и он также очень быстрый. Мы просто будем хранить адрес нужной процедуры в `EBX`, а затем выполним:

```
call    ebx
```

Это возможно быстрее, чем жёстко задавать адрес в коде, потому что микропроцессору не нужно извлекать адрес из памяти — он уже хранится в одном из его регистров. Я сказал *возможно*, потому что с учётом кэширования, которое выполняют современные микропроцессоры, оба варианта могут быть одинаково быстрыми.

# 11.3. Отображенные в память файлы

Поскольку наша программа работает с одним файлом, мы не можем использовать подход, который работал ранее, то есть чтение из входного файла и запись в выходной файл.

UNIX® позволяет нам отображать файл или его часть в память. Для этого сначала необходимо открыть файл с соответствующими флагами чтения/записи. Затем мы используем системный вызов `mmap`, чтобы отобразить его в память. Одно из преимуществ `mmap` заключается в том, что он автоматически работает с виртуальной памятью: мы можем отобразить в память больше файла, чем имеется физической памяти, и при этом обращаться к нему с помощью обычных команд работы с памятью, таких как `mov`, `lods` и `stos`. Все изменения, внесённые в память, отображённую из файла, будут записаны в файл системой. Нам даже не нужно держать файл открытым: пока он остаётся отображённым, мы можем читать из него и записывать в него.

32-разрядные микропроцессоры Intel могут адресовать до четырёх гигабайт памяти — физической или виртуальной. Система FreeBSD позволяет использовать до половины этого объёма для отображения файлов.

Для упрощения в этом руководстве мы будем преобразовывать только файлы, которые могут быть полностью отображены в памяти. Вероятно, не так много текстовых файлов превышают размер в два гигабайта. Если наша программа встретит такой файл, она просто выведет сообщение с предложением использовать оригинальный tuc.

Если вы изучите свою копию файла syscalls.master, вы найдёте два отдельных системных вызова с именем `mmap`. Это связано с эволюцией UNIX®: существовал традиционный BSD `mmap`, системный вызов 71. Он был заменён на POSIX® `mmap`, системный вызов 197. Система FreeBSD поддерживает оба, поскольку старые программы были написаны с использованием оригинальной BSD-версии. Но новое программное обеспечение использует версию POSIX®, которую мы и будем применять.

В syscalls.master POSIX® версия указана следующим образом:

```
197 STD BSD { caddr_t mmap(caddr_t addr, size_t len, int prot, \
                int flags, int fd, long pad, off_t pos); }
```

Это немного отличается от того, что указано в mmap(2). Это связано с тем, что mmap(2) описывает версию на языке C.

Разница заключается в аргументе `long pad`, который отсутствует в версии на C. Однако системные вызовы FreeBSD добавляют 32-битный заполнитель после `push` 64-битного аргумента. В данном случае `off_t` является 64-битным значением.

Когда мы завершаем работу с файлом, отображённым в память, мы освобождаем его с помощью системного вызова `munmap`:



Для подробного изучения `mmap` см. Unix Network Programming, Volume 2, Chapter 12 У. Ричарда Стивенса.

## 11.4. Определение размера файла

Поскольку нам нужно указать `mmap`, сколько байт файла отобразить в памяти, и поскольку мы хотим отобразить весь файл, нам необходимо определить его размер.

Мы можем использовать системный вызов `fstat` для получения всей информации об открытом файле, которую система может нам предоставить. Это включает в себя размер файла.

Вновь, в syscalls.master указаны две версии `fstat`: традиционная (системный вызов 62) и POSIX® (системный вызов 189). Естественно, мы будем использовать версию POSIX®:

```
189 STD POSIX   { int fstat(int fd, struct stat *sb); }
```

Это очень простой вызов: мы передаем ему адрес структуры `stat` и дескриптор открытого файла. Он заполнит содержимое структуры `stat`.

Однако должен сказать, что я пытался объявить структуру `stat` в секции `.bss`, и `fstat` это не понравилось: был установлен флаг переноса, указывающий на ошибку. После того как я изменил код, чтобы разместить структуру в стеке, всё заработало как надо.

## 11.5. Изменение размера файла

Поскольку наша программа может объединять последовательности возврата каретки / перевода строки в простые переводы строк, наш вывод может быть меньше, чем ввод. Однако, так как мы помещаем вывод в тот же файл, из которого читаем ввод, нам может потребоваться изменить размер файла.

Системный вызов `ftruncate` позволяет нам сделать именно это. Несмотря на название , несколько вводящее в заблуждение, системный вызов `ftruncate` может использоваться как для усечения файла (уменьшения его размера), так и для его увеличения.

И да, мы найдем две версии `ftruncate` в syscalls.master, старую (130) и новую (201). Мы будем использовать новую:

```
201 STD BSD { int ftruncate(int fd, int pad, off_t length); }
```

Обратите внимание, что здесь снова присутствует `int pad`.

## 11.6. ftuc

Теперь мы знаем всё, что нужно для написания ftuc. Начнём с добавления нескольких новых строк в system.inc. Сначала определим некоторые константы и структуры, где-нибудь в начале или около начала файла:

```
;;;;;;; open flags
```

```
%define O_RDONLY    0
%define O_WRONLY    1
%define O_RDWR  2

;;;;;;; mmap flags
%define PROT_NONE   0
%define PROT_READ   1
%define PROT_WRITE  2
%define PROT_EXEC   4
;;
%define MAP_SHARED  0001h
%define MAP_PRIVATE 0002h

;;;;;;; stat structure
struc   stat
st_dev      resd    1   ; = 0
st_ino      resd    1   ; = 4
st_mode     resw    1   ; = 8, size is 16 bits
st_nlink    resw    1   ; = 10, ditto
st_uid      resd    1   ; = 12
st_gid      resd    1   ; = 16
st_rdev     resd    1   ; = 20
st_atime    resd    1   ; = 24
st_atimensec    resd    1   ; = 28
st_mtime    resd    1   ; = 32
st_mtimensec    resd    1   ; = 36
st_ctime    resd    1   ; = 40
st_ctimensec    resd    1   ; = 44
st_size     resd    2   ; = 48, size is 64 bits
st_blocks   resd    2   ; = 56, ditto
st_blksize  resd    1   ; = 64
st_flags    resd    1   ; = 68
st_gen      resd    1   ; = 72
st_lspare   resd    1   ; = 76
st_qspare   resd    4   ; = 80
endstruc
```

Мы определяем новые системные вызовы:

```
%define SYS_mmap    197
%define SYS_munmap  73
%define SYS_fstat   189
%define SYS_ftruncate   201
```

Добавляем макросы для их использования:

```
%macro  sys.mmap    0
    system  SYS_mmap
%endmacro
```

```
%macro  sys.munmap  0
    system  SYS_munmap
%endmacro

%macro  sys.ftruncate   0
    system  SYS_ftruncate
%endmacro

%macro  sys.fstat   0
    system  SYS_fstat
%endmacro
```

И вот наш код:

```
;;;;;;; Fast Text-to-Unix Conversion (ftuc.asm) ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Started: 21-Dec-2000
;; Updated: 22-Dec-2000
;;
;; Copyright 2000 G. Adam Stanislav.
;; All rights reserved.
;;
;;;;;;; v.1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include    'system.inc'

section .data
    db  'Copyright 2000 G. Adam Stanislav.', 0Ah
    db  'All rights reserved.', 0Ah
usg db  'Usage: ftuc filename', 0Ah
usglen  equ $-usg
co  db  "ftuc: Can't open file.", 0Ah
colen   equ $-co
fae db  'ftuc: File access error.', 0Ah
faelen  equ $-fae
ftl db  'ftuc: File too long, use regular tuc instead.', 0Ah
ftllen  equ $-ftl
mae db  'ftuc: Memory allocation error.', 0Ah
maelen  equ $-mae

section .text

align 4
memerr:
    push    dword maelen
    push    dword mae
    jmp short error

align 4
toolong:
```

```
    push    dword ftllen
    push    dword ftl
    jmp short error

align 4
facerr:
    push    dword faelen
    push    dword fae
    jmp short error

align 4
cantopen:
    push    dword colen
    push    dword co
    jmp short error

align 4
usage:
    push    dword usglen
    push    dword usg

error:
    push    dword stderr
    sys.write

    push    dword 1
    sys.exit

align 4
global  _start
_start:
    pop eax     ; argc
    pop eax     ; program name
    pop ecx     ; file to convert
    jecxz   usage

    pop eax
    or  eax, eax    ; Too many arguments?
    jne usage

    ; Open the file
    push    dword O_RDWR
    push    ecx
    sys.open
    jc  cantopen

    mov ebp, eax    ; Save fd

    sub esp, byte stat_size
    mov ebx, esp
```

```
    ; Find file size
    push    ebx
    push    ebp     ; fd
    sys.fstat
    jc  facerr

    mov edx, [ebx + st_size + 4]

    ; File is too long if EDX != 0 ...
    or  edx, edx
    jne near toolong
    mov ecx, [ebx + st_size]
    ; ... or if it is above 2 GB
    or  ecx, ecx
    js  near toolong

    ; Do nothing if the file is 0 bytes in size
    jecxz   .quit

    ; Map the entire file in memory
    push    edx
    push    edx     ; starting at offset 0
    push    edx     ; pad
    push    ebp     ; fd
    push    dword MAP_SHARED
    push    dword PROT_READ | PROT_WRITE
    push    ecx     ; entire file size
    push    edx     ; let system decide on the address
    sys.mmap
    jc  near memerr

    mov edi, eax
    mov esi, eax
    push    ecx     ; for SYS_munmap
    push    edi

    ; Use EBX for state machine
    mov ebx, ordinary
    mov ah, 0Ah
    cld

.loop:
    lodsb
    call    ebx
    loop    .loop

    cmp ebx, ordinary
    je  .filesize

    ; Output final lf
    mov al, ah
```

```
    stosb
    inc edx

.filesize:
    ; truncate file to new size
    push    dword 0     ; high dword
    push    edx     ; low dword
    push    eax     ; pad
    push    ebp
    sys.ftruncate

    ; close it (ebp still pushed)
    sys.close

    add esp, byte 16
    sys.munmap

.quit:
    push    dword 0
    sys.exit

align 4
ordinary:
    cmp al, 0Dh
    je  .cr

    cmp al, ah
    je  .lf

    stosb
    inc edx
    ret

align 4
.cr:
    mov ebx, cr
    ret

align 4
.lf:
    mov ebx, lf
    ret

align 4
cr:
    cmp al, 0Dh
    je  .cr

    cmp al, ah
    je  .lf
```

```
    xchg    al, ah
    stosb
    inc edx

    xchg    al, ah
    ; fall through

.lf:
    stosb
    inc edx
    mov ebx, ordinary
    ret

align 4
.cr:
    mov al, ah
    stosb
    inc edx
    ret

align 4
lf:
    cmp al, ah
    je  .lf

    cmp al, 0Dh
    je  .cr

    xchg    al, ah
    stosb
    inc edx

    xchg    al, ah
    stosb
    inc edx
    mov ebx, ordinary
    ret

align 4
.cr:
    mov ebx, ordinary
    mov al, ah
    ; fall through

.lf:
    stosb
    inc edx
    ret
```



Не используйте эту программу для файлов, хранящихся на диске,

> отформатированном в MS-DOS® или Windows®. В коде FreeBSD присутствует неочевидная ошибка при использовании `mmap` на таких дисках, смонтированных в FreeBSD: если размер файла превышает определённое значение, `mmap` заполнит память нулями, а затем запишет их в файл, перезаписав его содержимое.

# 12. Спокойствие ума

Как ученик дзэн, мне нравится идея спокойствия ума (экаггата): делай одно дело за раз и делай его хорошо.

Вот именно так, в большинстве случаев, работает и UNIX®. В то время как типичное приложение Windows® пытается сделать всё, что только можно (и поэтому кишит ошибками), типичная программа UNIX® делает только одну вещь, но делает её хорошо.

Типичный пользователь UNIX® по сути собирает свои собственные приложения, написав shell-скрипт, который объединяет различные существующие программы, передавая вывод одной программы на вход другой.

При написании собственного программного обеспечения для UNIX® обычно рекомендуется определить, какие части решаемой задачи могут быть обработаны существующими программами, и создавать собственные программы только для той части задачи, для которой нет готового решения.

## 12.1. CSV

Я проиллюстрирую этот принцип конкретным примером из реальной жизни, с которым недавно столкнулся:

Мне нужно было извлечь 11-е поле каждой записи из базы данных, которую я загрузил с веб-сайта. База данных представляла собой CSV-файл, то есть список *значений, разделённых запятыми*. Это довольно стандартный формат для обмена данными между людьми, которые могут использовать разное программное обеспечение для работы с базами данных.

Первая строка файла содержит список различных полей, разделенных запятыми. Остальная часть файла содержит данные, перечисленные построчно, со значениями, разделенными запятыми.

Я попробовал awk, используя запятую в качестве разделителя. Но поскольку несколько строк содержали запятую в кавычках, awk извлекал неправильное поле из этих строк.

Следовательно, мне нужно было написать собственное программное обеспечение для извлечения 11-го поля из CSV-файла. Однако, следуя духу UNIX®, мне нужно было лишь создать простой фильтр, выполняющий следующие действия:

- Удалить первую строку из файла;
- Заменить все не заключённые в кавычки запятые на другой символ;
- Удалить все кавычки.

Строго говоря, я мог бы использовать sed для удаления первой строки из файла, но сделать это в моей собственной программе было очень просто, поэтому я решил так поступить и уменьшить размер конвейера.

В любом случае, написание подобной программы заняло у меня около 20 минут. Написание программы, которая извлекает 11-е поле из CSV-файла, заняло бы гораздо больше времени, и я не смог бы повторно использовать её для извлечения другого поля из другой базы данных.

На этот раз я решил позволить ей выполнить немного больше работы, чем обычная учебная программа:

- Она анализирует свою командную строку на наличие опций;
- Она отображает подсказку, если обнаруживает неверные аргументы;
- Она выдаёт понятные сообщения об ошибках.

Вот какое сообщение она выводит о том, как её использовать:

```
Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]
```

Все параметры необязательны и могут располагаться в любом порядке.

Параметр `-t` указывает, на что заменить запятые. По умолчанию используется `tab`. Например, `-t;` заменит все незакавыченные запятые на точку с запятой.

Мне не понадобилась опция `-c`, но в будущем она может пригодиться. Она позволяет указать, что я хочу заменить символ, отличный от запятой, на что-то другое. Например, `-c@` заменит все знаки @ (полезно, если нужно разделить список email-адресов на имена пользователей и домены).

Опция `-p` сохраняет первую строку, т.е. не удаляет её. По умолчанию мы удаляем первую строку, потому что в CSV-файле она содержит названия полей, а не данные.

Опции `-i` и `-o` позволяют указать входной и выходной файлы. По умолчанию используются stdin и stdout, как обычно работает стандартный фильтр UNIX®.

Я убедился, что принимаются как `-i filename`, так и `-ifilename`. Также я убедился, что может быть указан только один входной и один выходной файл.

Чтобы получить 11-е поле каждой записи, теперь я могу сделать:

```
% csv '-t;' data.csv | awk '-F;' '{print $11}'
```

Код сохраняет параметры (за исключением файловых дескрипторов) в `EDX`: запятая в `DH`, новый разделитель в `DL`, а флаг параметра `-p` в старшем бите `EDX`, поэтому проверка его знака даст нам быстрое решение о дальнейших действиях.

Вот код:

```
;;;;;;; csv.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Convert a comma-separated file to a something-else separated file.
;
; Started:  31-May-2001
; Updated:   1-Jun-2001
;
; Copyright (c) 2001 G. Adam Stanislav
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include    'system.inc'

%define BUFSIZE 2048

section .data
fd.in   dd  stdin
fd.out  dd  stdout
usg db  'Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]', 0Ah
usglen  equ $-usg
iemsg   db  "csv: Can't open input file", 0Ah
iemlen  equ $-iemsg
oemsg   db  "csv: Can't create output file", 0Ah
oemlen  equ $-oemsg

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
align 4
ierr:
    push    dword iemlen
    push    dword iemsg
    push    dword stderr
    sys.write
    push    dword 1     ; return failure
    sys.exit

align 4
oerr:
    push    dword oemlen
    push    dword oemsg
    push    dword stderr
    sys.write
    push    dword 2
    sys.exit

align 4
```

```
usage:
    push    dword usglen
    push    dword usg
    push    dword stderr
    sys.write
    push    dword 3
    sys.exit

align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]
    mov edx, (',' << 8) | 9

.arg:
    pop ecx
    or  ecx, ecx
    je  near .init      ; no more arguments

    ; ECX contains the pointer to an argument
    cmp byte [ecx], '-'
    jne usage

    inc ecx
    mov ax, [ecx]

.o:
    cmp al, 'o'
    jne .i

    ; Make sure we are not asked for the output file twice
    cmp dword [fd.out], stdout
    jne usage

    ; Find the path to output file - it is either at [ECX+1],
    ; i.e., -ofile --
    ; or in the next argument,
    ; i.e., -o file

    inc ecx
    or  ah, ah
    jne .openoutput
    pop ecx
    jecxz   usage

.openoutput:
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
```

```
    jc  near oerr

    add esp, byte 12
    mov [fd.out], eax
    jmp short .arg

.i:
    cmp al, 'i'
    jne .p

    ; Make sure we are not asked twice
    cmp dword [fd.in], stdin
    jne near usage

    ; Find the path to the input file
    inc ecx
    or  ah, ah
    jne .openinput
    pop ecx
    or  ecx, ecx
    je near usage

.openinput:
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  near ierr       ; open failed

    add esp, byte 8
    mov [fd.in], eax
    jmp .arg

.p:
    cmp al, 'p'
    jne .t
    or  ah, ah
    jne near usage
    or  edx, 1 << 31
    jmp .arg

.t:
    cmp al, 't'     ; redefine output delimiter
    jne .c
    or  ah, ah
    je  near usage
    mov dl, ah
    jmp .arg

.c:
    cmp al, 'c'
    jne near usage
```

```
    or  ah, ah
    je  near usage
    mov dh, ah
    jmp .arg

align 4
.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

    ; See if we are to preserve the first line
    or  edx, edx
    js  .loop

.firstline:
    ; get rid of the first line
    call    getchar
    cmp al, 0Ah
    jne .firstline

.loop:
    ; read a byte from stdin
    call    getchar

    ; is it a comma (or whatever the user asked for)?
    cmp al, dh
    jne .quote

    ; Replace the comma with a tab (or whatever the user wants)
    mov al, dl

.put:
    call    putchar
    jmp short .loop

.quote:
    cmp al, '"'
    jne .put

    ; Print everything until you get another quote or EOL. If it
    ; is a quote, skip it. If it is EOL, print it.
.qloop:
    call    getchar
    cmp al, '"'
    je  .loop

    cmp al, 0Ah
    je  .put
```

```
    call    putchar
    jmp short .qloop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    jecxz   .read
    call    write

.read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close

    ; return success
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
```

```
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    jecxz   .ret    ; nothing to write
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
.ret:
    ret
```

Большая часть взята из hex.asm выше. Однако есть одно важное отличие: я больше не вызываю `write` каждый раз при выводе перевода строки. Тем не менее, код можно использовать интерактивно.

Я нашел лучшее решение для интерактивной проблемы с тех пор, как начал писать эту главу. Я хотел убедиться, что каждая строка выводится отдельно только при необходимости. В конце концов, нет необходимости выводить каждую строку при неинтерактивном использовании.

Новое решение, которое я использую сейчас, заключается в вызове `write` каждый раз, когда обнаруживаю, что входной буфер пуст. Таким образом, при работе в интерактивном режиме программа считывает одну строку с клавиатуры пользователя, обрабатывает её и видит, что входной буфер пуст. Она сбрасывает свой вывод и читает следующую строку.

### 12.1.1. Темная сторона буферизации

Это изменение предотвращает загадочную блокировку в очень специфическом случае. Я называю это *тёмной стороной буферизации*, в основном потому, что это представляет опасность, которая не совсем очевидна.

Маловероятно, что это произойдет с такой программой, как csv выше, поэтому рассмотрим ещё один фильтр: в этом случае мы ожидаем, что наши входные данные будут представлять собой необработанные данные, описывающие значения цветов, такие как интенсивности *красного*, *зеленого* и *синего* для пикселя. На выходе мы получим негатив входных данных.

Такой фильтр было бы очень просто написать. Большая его часть выглядела бы так же, как и все другие фильтры, которые мы уже писали, поэтому я покажу только его внутренний цикл:

```
.loop:
    call    getchar
```

```
    not al      ; Create a negative
    call    putchar
    jmp short .loop
```

Поскольку этот фильтр работает с необработанными данными, он вряд ли будет использоваться интерактивно.

Но он может вызываться программным обеспечением для обработки изображений. И, если он не вызывает `write` перед каждым вызовом `read`, высока вероятность, что он зависнет.

Вот что может произойти:

1. Редактор изображений загрузит наш фильтр, используя функцию `popen()` на языке C.
2. Он прочитает первый ряд пикселей из битовой карты или пиксельной карты.
3. Он запишет первую строку пикселей в *канал*, ведущий к `fd.in` нашего фильтра.
4. Наш фильтр будет читать каждый пиксель из входных данных, преобразовывать его в негатив и записывать в выходной буфер.
5. Наш фильтр будет вызывать `getchar` для получения следующего пикселя.
6. `getchar` обнаружит пустой входной буфер, поэтому вызовет `read`.
7. `read` вызовет системный вызов `SYS_read`.
8. *Ядро* приостановит работу нашего фильтра до тех пор, пока редактор изображений не отправит больше данных в канал.
9. Редактор изображений будет читать из другого канала, подключенного к `fd.out` нашего фильтра, чтобы он мог установить первую строку выходного изображения *до* того, как отправит нам вторую строку входного.
10. *Ядро* приостанавливает работу графического редактора до тех пор, пока не получит какие-либо данные от нашего фильтра, чтобы передать их редактору.

На этом этапе наш фильтр ожидает, что редактор изображений отправит ему больше данных для обработки, в то время как редактор изображений ожидает, что наш фильтр отправит ему результат обработки первой строки. Однако результат находится в нашем выходном буфере.

Фильтр и редактор изображений будут продолжать ждать друг друга вечно (или, по крайней мере, пока их не завершат командой kill). Наше программное обеспечение только что вошло в состояние гонки.

Эта проблема не возникает, если наш фильтр очищает свой выходной буфер *перед* запросом к *ядру* для получения дополнительных входных данных.

# 13. Использование FPU

Как ни странно, большая часть литературы по ассемблеру даже не упоминает о существовании FPU, или *блока обработки чисел с плавающей запятой*, не говоря уже о программировании для него.

Тем не менее, язык ассемблера проявляет себя наилучшим образом, когда мы создаем высокооптимизированный код для FPU, выполняя вещи, которые можно сделать *только* на языке ассемблера.

# 13.1. Организация FPU

FPU состоит из 8 80-битных регистров с плавающей запятой. Они организованы в виде стека — вы можете `push` (поместить) значение на TOS (*вершина стека*) и `pop` (извлечь) его.

Как бы то ни было, мнемоники ассемблера — не `push` и `pop`, потому что они уже заняты.

Вы можете `push` (положить) значение на вершину стека (TOS), используя `fld`, `fild` и `fbld`. Несколько других кодов операций позволяют вам `push` (положить) многие распространённые *константы* — например, *pi* — на вершину стека (TOS).

Аналогично, вы можете `извлечь` значение с помощью `fst`, `fstp`, `fist`, `fistp` и `fbstp`. На самом деле только коды операций, оканчивающиеся на *p*, буквально `извлекают` значение, остальные же `сохраняют` его в другом месте, не удаляя с вершины стека (TOS).

Мы можем передавать данные между TOS и памятью компьютера либо как 32-битное, 64-битное или 80-битное *вещественное* число, 16-битное, 32-битное или 64-битное *целое* число, либо как 80-битное *упакованное десятичное* число.

80-битный *упакованный десятичный* формат является особым случаем *двоично-десятичного кодирования*, который очень удобен при преобразовании между ASCII-представлением данных и внутренними данными FPU. Он позволяет использовать до 18 значащих цифр.

Независимо от того, как мы представляем данные в памяти, FPU всегда хранит их в 80-битном формате *real* в своих регистрах.

Его внутренняя точность составляет не менее 19 десятичных цифр, поэтому даже если мы решим отображать результаты в формате ASCII с полной 18-значной точностью, мы всё равно будем показывать корректные результаты.

Мы можем выполнять математические операции над TOS: вычислять его *синус*, *масштабировать* (то есть умножать или делить на степень двойки), вычислять его *логарифм* по основанию 2 и многое другое.

Мы также можем *умножить* или *разделить* его, *прибавить* к нему или *вычесть* его из любого из регистров FPU (включая его самого).

Официальный код операции Intel для TOS — `st`, а для *регистров* — `st(0)`-`st(7)`. Таким образом, `st` и `st(0)` ссылаются на один и тот же регистр.

По каким-то причинам оригинальный автор nasm решил использовать другие коды операций, а именно `st0`-`st7`. Другими словами, скобки отсутствуют, а вершина стека всегда `st0`, но никогда просто `st`.

### 13.1.1. Формат упакованного десятичного числа

Формат *упакованного десятичного числа* использует 10 байт (80 бит) памяти для представления 18 цифр. Представленное число всегда является *целым*.



Вы можете использовать это для получения десятичных знаков, предварительно умножив TOS на степень 10.

Старший бит старшего байта (байт 9) является *знаковым битом*: если он установлен, число *отрицательное*, в противном случае — *положительное*. Остальные биты этого байта не используются/игнорируются.

Оставшиеся 9 байт хранят 18 цифр числа: 2 цифры на байт.

*Старший разряд* хранится в старшем *полубайте* (4 бита), *младший разряд* — в младшем *полубайте*.

Как бы то ни было, вы можете подумать, что `-1234567` будет храниться в памяти следующим образом (в шестнадцатеричной записи):

```
80 00 00 00 00 00 01 23 45 67
```

Увы, это не так! Как и все остальное, созданное Intel, даже *упакованное десятичное число* имеет порядок *от младшего к старшему*.

Это означает, что наш `-1234567` хранится следующим образом:

```
67 45 23 01 00 00 00 00 00 80
```

Помните об этом, иначе вы будете рвать на себе волосы в отчаянии!



Книга, которую стоит прочитать — если сможете её найти — это книга Ричарда Старца 8087/80287/80387 для IBM PC и совместимых. Хотя в ней, кажется, факт о little-endian хранении *упакованного десятичного числа* принимается как данность. Я не шучу насчёт отчаяния, которое испытывал, пытаясь понять, что не так с фильтром, который я привожу ниже, *прежде* чем мне пришло в голову попробовать little-endian порядок даже для этого типа данных.

## 13.2. Экскурсия в фотографию с помощью камеры-обскуры

Чтобы создавать полезное программное обеспечение, мы должны понимать не только наши инструменты программирования, но и область, для которой разрабатываем ПО.

Наш следующий фильтр поможет нам, когда мы захотим создать *камеру-обскуру*, поэтому

нам понадобятся некоторые знания о *фотографии с помощью обскуры*, прежде чем мы сможем продолжить.

### 13.2.1. Камера

Самый простой способ описать любую когда-либо созданную камеру — это некоторое пустое пространство, заключённое в светонепроницаемый материал, с небольшим отверстием в корпусе.

Корпус обычно прочный (например, коробка), хотя иногда он гибкий (гофрированная часть). Внутри камеры довольно темно. Однако отверстие пропускает световые лучи через одну точку (хотя в некоторых случаях их может быть несколько). Эти световые лучи формируют изображение — представление того, что находится снаружи камеры, перед отверстием.

Если внутрь камеры поместить светочувствительный материал (например, плёнку), он может зафиксировать изображение.

Отверстие часто содержит *линзу* или сборку линз, которую часто называют *объективом*.

### 13.2.2. Игольное ушко

Но, строго говоря, линза не обязательна: первые камеры использовали не линзу, а *маленькое отверстие* размером с игольное ушко. Даже сегодня *маленькие отверстия* применяются как инструмент для изучения принципов работы камер и для создания особого вида изображений.

Изображение, создаваемое *маленьким отверстием*, одинаково резкое. Или *размытое*. Существует идеальный размер для маленького отверстия: если оно больше или меньше, изображение теряет резкость.

### 13.2.3. Фокусное расстояние

Идеальный диаметр отверстия является функцией квадратного корня из *фокусного расстояния*, которое представляет собой расстояние от отверстия до плёнки.

```
D = PC * sqrt(FL)
```

Здесь `D` — идеальный диаметр отверстия, `FL` — фокусное расстояние, а `PC` — константа отверстия. По данным Джейя Бендера, её значение равно `0,04`, тогда как Кеннет Коннорс определил его как `0,037`. Другие исследователи предложили иные значения. Кроме того, это значение справедливо только для дневного света: другие типы освещения потребуют иной константы, значение которой можно определить только экспериментальным путём.

### 13.2.4. Число f (диафрагменное число)

Число f — это очень полезный показатель того, сколько света попадает на плёнку. Экспонометр может определить, что, например, для экспонирования плёнки определённой

чувствительности при f/5.6 может потребоваться выдержка 1/1000 сек.

Не имеет значения, 35-мм это камера или камера 6x9 см и т.д. Достаточно знать диафрагменное число, чтобы определить правильную экспозицию.

Число f легко вычислить:

```
F = FL / D
```

Другими словами, число f равно фокусному расстоянию, деленному на диаметр отверстия. Это также означает, что большее f-число подразумевает либо меньшее отверстие, либо большее фокусное расстояние, либо и то, и другое. В свою очередь, это означает, что чем больше число f, тем дольше должна быть выдержка.

Кроме того, хотя диаметр отверстия и фокусное расстояние являются одномерными величинами, и плёнка, и отверстие — двумерны. Это означает, что если вы измерили экспозицию при диафрагменном числе `A` как `t`, то экспозиция при диафрагменном числе `B` будет:

```
t * (B / A)²
```

### 13.2.5. Нормализованное число f

Хотя многие современные камеры могут изменять диаметр своего отверстия, а следовательно и свое число f, довольно плавно и постепенно, так было не всегда.

Для обеспечения различных значений диафрагмы в камерах обычно использовалась металлическая пластина с несколькими отверстиями разного размера.

Их размеры были выбраны в соответствии с приведённой выше формулой таким образом, чтобы результирующее f-число было одним из стандартных f-чисел, используемых на всех фотоаппаратах. Например, у моего очень старого фотоаппарата Kodak Duaflex IV есть три таких отверстия для чисел f — 8, 11 и 16.

Более современные камеры могут предлагать значения диафрагменного числа 2.8, 4, 5.6, 8, 11, 16, 22 и 32 (а также другие). Эти числа выбраны не произвольно: все они являются степенями квадратного корня из 2, хотя могут быть немного округлены.

### 13.2.6. Ступени числа f

Типичная камера устроена так, что установка любого из нормализованных чисел f изменяет ощущение от регулятора. Он естественным образом *останавливается* в этом положении. Из-за этого такие положения регулятора называются f-ступенями.

Поскольку значения диафрагмы на каждой ступени являются степенями квадратного корня из 2, поворот диска на 1 ступень удваивает количество света, необходимое для правильной экспозиции. Поворот на 2 ступени увеличивает требуемую экспозицию вчетверо. Поворот диска на 3 ступени требует увеличения экспозиции в 8 раз и так далее.

# 13.3. Проектирование программного обеспечения камеры-обскуры

Мы готовы решить, что именно должно делать наше программное обеспечение для камер-обскур.

## 13.3.1. Обработка ввода программы

Поскольку основная цель — помочь нам разработать работающую камеру-обскуру, мы будем использовать *фокусное расстояние* в качестве входных данных для программы. Это можно определить без программного обеспечения: правильное фокусное расстояние зависит от размера плёнки и необходимости съёмки «обычных» изображений, широкоугольных или телефото.

Большинство написанных нами до сих пор программ работали с отдельными символами или байтами в качестве входных данных: программа hex преобразовывала отдельные байты в шестнадцатеричное число, программа csv либо пропускала символ, либо удаляла его, либо заменяла на другой символ и т.д.

Одна программа, `ftuc`, использовала конечный автомат для обработки не более двух входных байтов за раз.

Но наша программа для камеры-обскуры не может работать только с отдельными символами, ей приходится иметь дело с более крупными синтаксическими единицами.

Например, если мы хотим, чтобы программа рассчитала диаметр отверстия (и другие значения, которые мы обсудим позже) для фокусных расстояний `100 мм`, `150 мм` и `210 мм`, мы можем ввести что-то вроде этого:

```
100, 150, 210
```

Наша программа должна учитывать более одного байта входных данных за раз. Когда она видит первую `1`, она должна понимать, что это первая цифра десятичного числа. Когда она видит `0` и другой `0`, она должна знать, что это следующие цифры того же числа.

Когда он встречает первую запятую, он должен понять, что больше не получает цифры первого числа. Он должен уметь преобразовать цифры первого числа в значение `100`. И цифры второго числа в значение `150`. И, конечно же, цифры третьего числа в числовое значение `210`.

Нам нужно определиться с допустимыми разделителями: должны ли входные числа разделяться запятой? Если да, то как обрабатывать два числа, разделённые чем-то другим?

Лично я предпочитаю простоту. Либо что-то является числом — и тогда я его обрабатываю. Либо не является числом — и тогда я это отбрасываю. Мне не нравится, когда компьютер жалуется на лишний символ, если *очевидно*, что он лишний. Да ладно!

Плюс, это позволяет мне разбавить монотонность вычислений и ввести запрос вместо

просто числа:

```
What is the best pinhole diameter for the
        focal length of 150?
```

Нет причины, чтобы компьютер выводил множество жалоб:

```
Syntax error: What
Syntax error: is
Syntax error: the
Syntax error: best
```

И так далее, и так далее, и так далее.

Во-вторых, мне нравится символ `#` для обозначения начала комментария, который продолжается до конца строки. Это не требует больших усилий для реализации и позволяет мне рассматривать входные файлы для моего программного обеспечения как исполняемые скрипты.

В нашем случае также необходимо определиться с единицами измерения входных данных: мы выбираем *миллиметры*, так как большинство фотографов измеряют фокусное расстояние именно в них.

Наконец, нам нужно решить, разрешать ли использование десятичной точки (в этом случае мы также должны учитывать тот факт, что во многих странах используется десятичная *запятая*).

В нашем случае разрешение десятичной точки/запятой создало бы ложное ощущение точности: разница между фокусными расстояниями `50` и `51` практически незаметна, поэтому разрешать пользователю вводить что-то вроде `50.5` — не лучшая идея. Это моё мнение, конечно, но программу пишу я. В своей программе вы можете сделать другие выбор, разумеется.

### 13.3.2. Передача параметров программе

Самое важное, что нам нужно знать при создании камеры-обскуры — это диаметр отверстия. Поскольку мы хотим получать чёткие изображения, мы будем использовать приведённую выше формулу для расчёта диаметра отверстия от фокусного расстояния. Поскольку эксперты предлагают несколько различных значений для константы `PC`, нам нужно будет иметь выбор.

В традициях программирования в UNIX® предусмотрены два основных способа выбора параметров программы, а также значение по умолчанию на случай, если пользователь не сделает выбор.

Почему есть два способа выбора?

Один из способов — это позволить (относительно) *постоянный* выбор, который

применяется автоматически каждый раз при запуске программы, без необходимости каждый раз указывать, что мы хотим, чтобы она сделала.

Постоянные настройки могут быть сохранены в конфигурационном файле, обычно расположенном в домашнем каталоге пользователя. Файл обычно имеет то же имя, что и приложение, но начинается с точки. Часто к имени файла добавляется *"rc"*. Таким образом, наш файл может называться ~/.pinhole или ~/.pinholerc. (Обозначение ~/ означает домашний каталог текущего пользователя.)

Файл конфигурации в основном используется программами, у которых много настраиваемых параметров. Те, у которых он один (или несколько), часто используют другой метод: они ожидают найти параметр в *переменной окружения*. В нашем случае, мы можем посмотреть на переменную окружения с именем `PINHOLE`.

Обычно программа использует один из вышеуказанных методов. В противном случае, если в конфигурационном файле указано одно, а в переменной окружения — другое, программа может запутаться (или стать слишком сложной).

Поскольку нам нужно выбрать только *один* такой параметр, мы воспользуемся вторым методом и поищем в окружении переменную с именем `PINHOLE`.

Другой способ позволяет нам принимать *ad hoc* решения: *"Хотя обычно я хочу, чтобы ты использовал 0.039, на этот раз мне нужно 0.03872."* Другими словами, он позволяет нам *переопределить* постоянный выбор.

Такой выбор обычно осуществляется с помощью параметров командной строки.

Наконец, программе *всегда* необходим *значение по умолчанию*. Пользователь может не делать никакого выбора. Возможно, он не знает, что выбрать. Возможно, он «просто просматривает». Предпочтительно, чтобы значением по умолчанию было то, что выбрало бы большинство пользователей. Таким образом, им не нужно выбирать. Или, точнее, они могут выбрать значение по умолчанию без дополнительных усилий.

Учитывая эту систему, программа может обнаружить конфликтующие параметры и обработать их следующим образом:

1. Если она находит *специальный* выбор (например, параметр командной строки), она должна принять этот выбор. Она должна игнорировать любой постоянный выбор и значения по умолчанию.
2. *В противном случае*, если будет найден постоянный параметр (например, переменная окружения), он должен быть принят, а значение по умолчанию — проигнорировано.
3. *В противном случае*, следует использовать значение по умолчанию.

Нам также необходимо решить, в каком *формате* должна быть наша опция `PC`.

На первый взгляд кажется очевидным использовать формат `PINHOLE=0.04` для переменной окружения и `-p0.04` для командной строки.

Разрешение этого на самом деле представляет угрозу безопасности. Константа `PC` — это очень маленькое число. Естественно, мы протестируем наше программное обеспечение,

используя различные небольшие значения `PC`. Но что произойдёт, если кто-то запустит программу, выбрав огромное значение?

Это может привести к сбою программы, так как мы не разрабатывали её для обработки огромных чисел.

Или мы можем потратить больше времени на программу, чтобы она могла обрабатывать огромные числа. Мы могли бы сделать это, если бы писали коммерческое программное обеспечение для аудитории, не знакомой с компьютерами.

Или можно сказать: *"Пусть терпит! Пользователь сам должен был разобраться."*

Или мы можем просто сделать невозможным ввод пользователем слишком большого числа. Это подход, который мы выберем: мы будем использовать *подразумеваемый префикс 0.*.

Другими словами, если пользователь хочет `0.04`, мы ожидаем, что он введёт `-p04` или установит `PINHOLE=04` в своём окружении. Таким образом, если он укажет `-p9999999`, мы интерпретируем это как `0.9999999` — всё ещё нелепо, но по крайней мере безопаснее.

Во-вторых, многие пользователи просто захотят использовать либо константу Бендера, либо константу Коннорса. Чтобы облегчить им задачу, мы будем интерпретировать `-b` как идентичное `-p04`, а `-c` как идентичное `-p037`.

### 13.3.3. Вывод результата

Нам нужно решить, что наше программное обеспечение должно отправлять на вывод и в каком формате.

Поскольку наши входные данные допускают неограниченное количество значений фокусного расстояния, имеет смысл использовать традиционный вывод в стиле базы данных, показывая результат вычислений для каждого фокусного расстояния на отдельной строке, разделяя все значения в строке символом табуляции.

Опционально, мы также должны разрешить пользователю указать использование формата CSV, который мы изучили ранее. В этом случае мы выведем строку с разделёнными запятыми названиями, описывающими каждое поле каждой строки, а затем отобразим результаты как прежде, но заменив `табуляцию` на `запятую`.

Нам нужна опция командной строки для формата CSV. Мы не можем использовать `-c`, потому что это уже означает *использовать константу Коннорса*. По какой-то странной причине многие веб-сайты называют CSV-файлы *"электронными таблицами Excel"* (хотя формат CSV появился раньше Excel). Поэтому мы будем использовать переключатель `-e`, чтобы указать нашему программному обеспечению, что мы хотим получить вывод в формате CSV.

Мы начнем каждую строку вывода с фокусного расстояния. Это может показаться избыточным сначала, особенно в интерактивном режиме: пользователь вводит фокусное расстояние, а мы его повторяем.

Но пользователь может ввести несколько фокусных расстояний в одной строке. Ввод также может поступать из файла или вывода другой программы. В этом случае пользователь вообще не видит вводимые данные.

Таким же образом, вывод может быть направлен в файл, который мы захотим изучить позже, или на принтер, или стать входными данными для другой программы.

Итак, имеет полный смысл начинать каждую строку с фокусного расстояния, введённого пользователем.

Нет, подождите! Не так, как ввел пользователь. Что, если пользователь введет что-то вроде этого:

```
00000000150
```

Очевидно, нам нужно удалить ведущие нули.

Итак, можно рассмотреть вариант чтения пользовательского ввода как есть, преобразования его в бинарный вид внутри FPU и последующего вывода оттуда.

Но…

Что делать, если пользователь введёт что-то вроде этого:

```
1745976572345235345353453535353053053456350730967676442 3
```

Ха! Упакованный десятичный формат FPU позволяет нам вводить 18-значные числа. Но пользователь ввёл больше 18 цифр. Как нам обработать это?

Хорошо, мы *могли бы* изменить наш код, чтобы он читал первые 18 цифр, передавал их в FPU, затем читал ещё, умножал уже имеющееся на вершине стека (TOS) на 10 в степени количества дополнительных цифр, а затем выполнял сложение с этим значением.

Да, мы могли бы так поступить. Но в *этой* программе это было бы нелепо (в другой это могло бы быть как раз тем, что нужно): даже длина окружности Земли, выраженная в миллиметрах, занимает всего 11 цифр. Очевидно, мы не можем построить камеру такого размера (по крайней мере, пока).

Итак, если пользователь вводит такое огромное число, он либо скучает, либо проверяет нас, либо пытается взломать систему, либо играет — делает что угодно, кроме проектирования камеры-обскуры.

Что мы будем делать?

Мы ударим его по лицу, образно говоря:

```
1745976572345235345353453535353053053456350730967676442 3    ??? ??? ??? ??? ???
```

Для этого мы просто проигнорируем все ведущие нули. Как только мы найдем ненулевую цифру, мы инициализируем счетчик значением `0` и начнем выполнять три шага:

1. Отправить цифру на выход.
2. Добавить цифру в буфер, который мы позже используем для создания упакованного десятичного числа, которое можно отправить в FPU.
3. Увеличить счетчик.

Теперь, пока мы выполняем эти три шага, нам также необходимо следить за одним из двух условий:

- Если счётчик превышает 18, мы прекращаем добавление в буфер. Мы продолжаем читать цифры и отправлять их на вывод.
- Если, или скорее *когда*, следующий вводимый символ не является цифрой, мы завершаем ввод на данный момент.

  Между прочим, мы можем просто отбросить нецифровой символ, если это не `#`, который необходимо вернуть во входной поток. Он начинает комментарий, поэтому мы должны увидеть его после завершения вывода и начала поиска следующего ввода.

Остается одна непокрытая возможность: если пользователь вводит только ноль (или несколько нулей), мы никогда не найдем ненулевое значение для отображения.

Мы можем определить, что это произошло, когда наш счетчик остаётся на `0`. В этом случае нам нужно отправить `0` на выход и выполнить ещё один "удар по лицу":

```
0   ??? ??? ??? ??? ???
```

Как только мы определили фокусное расстояние и убедились, что оно корректно (больше `0`, но не превышает 18 цифр), можно рассчитать диаметр отверстия.

Не случайно слово *булавочное ушко* содержит слово *булавка*. Действительно, многие малые отверстия буквально являются *дырками от булавки* — отверстиями, аккуратно проделанными остриём булавки.

Вот потому что типичное отверстие очень маленькое. Наша формула даёт результат в миллиметрах. Мы умножим его на `1000`, чтобы вывести результат в *микронах*.

На этом этапе нас ожидает ещё одна ловушка: *Излишняя точность*.

Да, FPU был разработан для вычислений с высокой точностью. Но мы имеем дело не с вычислениями высокой точности. Мы имеем дело с физикой (конкретно, с оптикой).

Предположим, мы хотим превратить грузовик в камеру-обскуру (мы будем не первыми, кто это сделал!). Допустим, его кузов имеет длину `12` метров, значит, фокусное расстояние равно `12000`. Используя константу Бендера, получаем квадратный корень из `12000`, умноженный на `0.04`, что составляет `4.381780460` миллиметра или `4381.780460` микрона.

Как ни посмотри, результат абсурдно точен. Наш грузовик не имеет *точно* `12000` миллиметров в длину. Мы не измеряли его длину с такой точностью, поэтому утверждение, что нам нужна отверстие диаметром `4,381780460` миллиметра, мягко говоря, вводит в заблуждение. `4,4` миллиметра будет вполне достаточно.



Я "всего лишь" использовал десять цифр в приведённом выше примере. Представьте абсурдность попытки использовать все 18!

Нам нужно ограничить количество значащих цифр в нашем результате. Один из способов сделать это — использовать целое число, представляющее микроны. Таким образом, нашему грузовику потребуется отверстие диаметром `4382` микрона. Глядя на это число, мы всё же решаем, что `4400` микрон, или `4.4` миллиметра, достаточно близко.

Кроме того, мы можем решить, что независимо от размера результата, мы хотим отображать только четыре значащих цифры (или любое другое их количество, конечно). Увы, FPU не поддерживает округление до определённого количества цифр (в конце концов, он воспринимает числа не как десятичные, а как двоичные).

Следовательно, мы должны разработать алгоритм для уменьшения количества значащих цифр.

Вот мой (я думаю, он неуклюжий — если у вас есть вариант лучше, *пожалуйста*, дайте мне знать):

1. Инициализировать счетчик значением `0`.
2. Пока число больше или равно `10000`, делим его на `10` и увеличиваем счётчик.
3. Вывести результат.
4. Пока счетчик больше `0`, выводить `0` и уменьшать счетчик.



`10000` подходит только если вам нужно *четыре* значащих цифры. Для любого другого количества значащих цифр замените `10000` на `10` в степени, равной количеству значащих цифр.

Мы затем выведем диаметр отверстия в микронах, округлённый до четырёх значащих цифр.

На этом этапе нам известны *фокусное расстояние* и *диаметр отверстия*. Это означает, что у нас достаточно информации для расчёта *диафрагменного числа*.

Мы отобразим число f, округлённое до четырёх значащих цифр. Скорее всего, само число f мало что нам скажет. Чтобы придать ему больше смысла, мы можем найти ближайшее *нормализованное число f*, то есть ближайшую степень квадратного корня из 2.

Мы делаем это, умножая фактическое значение диафрагмы на само себя, что, конечно же, даст нам его `квадрат`. Затем мы вычислим его логарифм по основанию 2, что намного проще, чем вычисление логарифма по основанию квадратного корня из 2! Мы округлим результат до ближайшего целого числа. Далее мы возведём 2 в полученную степень. На самом деле, FPU предоставляет нам удобный способ сделать это: мы можем использовать код операции

`fscale` для "масштабирования" 1, что аналогично `сдвигу` целого числа влево. Наконец, мы вычисляем квадратный корень из всего этого и получаем ближайшее нормализованное значение диафрагмы.

Если всё это звучит ошеломляюще — или, возможно, слишком сложно — всё может стать гораздо понятнее, если увидеть код. Вместе это занимает всего 9 инструкций процессора:

```
fmul    st0, st0
    fld1
    fld st1
    fyl2x
    frndint
    fld1
    fscale
    fsqrt
    fstp    st1
```

Первая строка, `fmul st0, st0`, возводит в квадрат содержимое TOS (вершина стека, то же что `st`, называется `st0` в nasm). Команда `fld1` помещает `1` на вершину стека.

Следующая строка, `fld st1`, помещает квадрат обратно в TOS. На этом этапе квадрат находится и в `st`, и в `st(2)` (скоро станет ясно, зачем мы оставляем вторую копию в стеке). В `st(1)` содержится `1`.

Далее, `fyl2x` вычисляет логарифм по основанию 2 от `st`, умноженный на `st(1)`. Именно поэтому мы ранее поместили `1` в `st(1)`.

На этом этапе `st` содержит логарифм, который мы только что вычислили, а `st(1)` содержит квадрат фактического значения диафрагменного числа, который мы сохранили для последующего использования.

`frndint` округляет TOS до ближайшего целого числа. `fld1` помещает `1` в стек. `fscale` сдвигает `1`, находящееся на TOS, на значение в `st(1)`, фактически возводя 2 в степень `st(1)`.

Наконец, `fsqrt` вычисляет квадратный корень из результата, т.е. ближайшее нормализованное число f.

У нас теперь есть ближайшее нормализованное число f на вершине стека (TOS), округлённый до ближайшего целого двоичный логарифм в `st(1)` и квадрат фактического число f в `st(2)`. Мы сохраняем значение в `st(2)` для последующего использования.

Но нам больше не нужно содержимое `st(1)`. Последняя строка, `fstp st1`, помещает содержимое `st` в `st(1)` и выполняет извлечение. В результате, то, что было `st(1)`, теперь становится `st`, то, что было `st(2)`, теперь становится `st(1)`, и так далее. Новый `st` содержит нормализованное число f. Новый `st(1)` содержит квадрат фактического число f, который мы сохранили для потомков.

На этом этапе мы готовы вывести нормализованное число f. Поскольку оно нормализовано, мы не будем округлять его до четырёх значащих цифр, а отправим его с полной точностью.

Нормализованное диафрагменное число полезно, пока оно достаточно мало и может быть найдено на нашем экспонометре. В противном случае нам нужен другой метод определения правильной экспозиции.

Ранее мы вывели формулу для расчёта правильной экспозиции при произвольной диафрагме на основе измерений, сделанных при другой диафрагме.

Каждый экспонометр, который я когда-либо видел, может определить правильную экспозицию при f5.6. Поэтому мы рассчитаем *"множитель f5.6"*, то есть насколько нужно умножить экспозицию, измеренную при f5.6, чтобы определить правильную экспозицию для нашей камеры-обскуры.

Из приведённой формулы мы знаем, что этот коэффициент можно вычислить, разделив наше число f (фактическое, а не нормализованное) на `5.6` и возведя результат в квадрат.

Математически, деление квадрата нашего числа f на квадрат `5.6` даст нам тот же результат.

С вычислительной точки зрения, нам не нужно возводить в квадрат два числа, когда можно возвести только одно. Таким образом, первое решение на первый взгляд кажется лучше.

Но…

`5.6` — это *константа*. Нам не нужно заставлять наш FPU тратить драгоценные циклы. Мы можем просто указать ему разделить квадрат f-числа на то, чему равно $5.6^2$. Или мы можем разделить f-число на `5.6`, а затем возвести результат в квадрат. Теперь оба способа кажутся равнозначными.

Но они не такие!

Изучив принципы фотографии выше, мы помним, что `5.6` — это квадратный корень из 2, возведённый в пятую степень. Это *иррациональное* число. Квадрат этого числа *ровно* `32`.

`32` — это не просто целое число, это степень двойки. Нам не нужно делить квадрат числа f на `32`. Достаточно использовать `fscale` для сдвига вправо на пять позиций. В терминологии FPU это означает, что мы применим `fscale` со значением `st(1)` равным `-5`. Это *гораздо быстрее*, чем деление.

Итак, теперь стало ясно, зачем мы сохранили квадрат числа f на вершине стека FPU. Расчёт множителя для f5.6 — это самое простое вычисление во всей программе! Мы выведем его, округлив до четырёх значащих цифр.

Есть ещё одно полезное число, которое мы можем вычислить: количество ступеней, на которые наше значение диафрагмы отличается от f5.6. Это может помочь, если наше значение диафрагмы находится чуть за пределами диапазона нашего экспонометра, но у нас есть затвор, который позволяет устанавливать различные выдержки, и этот затвор использует ступени.

Предположим, наше число диафрагмы на 5 ступеней отличается от f5.6, а экспонометр показывает, что нужно использовать выдержку 1/1000 сек. Тогда мы можем сначала установить выдержку на 1/1000, а затем повернуть диск на 5 ступеней.

Этот расчет также довольно прост. Все, что нам нужно сделать, это вычислить логарифм по основанию 2 от множителя f5.6, который мы только что рассчитали (хотя нам нужно его значение до округления). Затем мы выводим результат, округленный до ближайшего целого числа. Нам не нужно беспокоиться о наличии более четырёх значащих цифр в этом случае: скорее всего, результат будет содержать только одну или две цифры.

## 13.4. Оптимизации FPU

В ассемблерном коде мы можем оптимизировать инструкции FPU способами, невозможными в языках высокого уровня, включая C.

Всякий раз, когда функции на языке C требуется вычислить значение с плавающей запятой, она загружает все необходимые переменные и константы в регистры FPU. Затем выполняются все необходимые вычисления для получения правильного результата. Хорошие компиляторы C могут очень эффективно оптимизировать эту часть кода.

Он "возвращает" значение, оставляя результат на вершине стека (TOS). Однако перед возвратом он выполняет очистку. Все переменные и константы, использованные в вычислениях, теперь удалены из FPU.

Он не может сделать то, что мы только что сделали выше: мы вычислили квадрат числа f и оставили его в стеке для последующего использования другой функцией.

Мы *знали*, что это значение понадобится позже. Мы также знали, что у нас достаточно места в стеке (в котором помещается только 8 чисел), чтобы сохранить его там.

Компилятор C не может знать, что значение, находящееся в стеке, потребуется снова в ближайшем будущем.

Конечно, программист на C может это знать. Но единственное средство, которое у него есть, — это сохранить значение в переменной памяти.

Это означает, что значение будет изменено с 80-битной точности, используемой внутри FPU, на тип *double* (64 бита) или даже *single* (32 бита) в C.

Это также означает, что значение должно быть перемещено из TOS в память, а затем обратно. Увы, среди всех операций с FPU, доступ к памяти компьютера является самым медленным.

Итак, при программировании FPU на языке ассемблера ищите способы хранения промежуточных результатов в стеке FPU.

Мы можем развить эту идею ещё дальше! В нашей программе мы используем *константу* (ту, которую назвали `PC`).

Не имеет значения, сколько диаметров отверстий мы рассчитываем: 1, 10, 20, 1000, мы всегда используем одну и ту же константу. Следовательно, мы можем оптимизировать нашу программу, сохраняя константу в стеке всё время.

В начале нашей программы мы вычисляем значение указанной константы. Нам нужно

разделить наш вход на 10 для каждой цифры в константе.

Гораздо быстрее умножать, чем делить. Поэтому в начале нашей программы мы делим 1 на 10, чтобы получить 0.1, который затем сохраняем в стеке: вместо того чтобы делить ввод на 10 для каждой цифры, мы умножаем его на 0.1.

Кстати, мы не вводим 0.1 напрямую, хотя могли бы. У нас есть причина для этого: хотя 0.1 можно выразить всего одним десятичным знаком, мы не знаем, сколько *двоичных* разрядов для этого потребуется. Поэтому мы позволяем FPU вычислить его двоичное значение с собственной высокой точностью.

Мы используем другие константы: умножаем диаметр отверстия на 1000, чтобы перевести его из миллиметров в микроны. Мы сравниваем числа с 10000, когда округляем их до четырёх значащих цифр. Таким образом, мы оставляем и 1000, и 10000 в стеке. И, конечно же, мы повторно используем 0.1 при округлении чисел до четырёх цифр.

И последнее, но не менее важное: мы оставляем -5 в стеке. Он нам нужен для масштабирования квадрата числа f вместо деления его на 32. Не случайно мы загружаем эту константу последней. Это делает её вершиной стека, когда в нём находятся только константы. Таким образом, при масштабировании квадрата число f -5 находится в st(1), именно там, где fscale ожидает его увидеть.

Это обычная ситуация, когда некоторые константы создаются с нуля, вместо загрузки их из памяти. Именно это мы делаем с -5:

```
    fld1            ; TOS =  1
    fadd    st0, st0    ; TOS =  2
    fadd    st0, st0    ; TOS =  4
    fld1            ; TOS =  1
    faddp   st1, st0    ; TOS =  5
    fchs            ; TOS = -5
```

Мы можем обобщить все эти оптимизации в одном правиле: *Держите повторяющиеся значения в стеке!*



*PostScript*® — это стековая язык программирования. Существует гораздо больше книг о PostScript®, чем о языке ассемблера FPU: освоение PostScript® поможет вам овладеть FPU.

## 13.5. Код pinhole

```
;;;;;;; pinhole.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Find various parameters of a pinhole camera construction and use
;
; Started:   9-Jun-2001
; Updated:  10-Jun-2001
```

```
;
; Copyright (c) 2001 G. Adam Stanislav
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include    'system.inc'

%define BUFSIZE 2048

section .data
align 4
ten dd  10
thousand    dd  1000
tthou   dd  10000
fd.in   dd  stdin
fd.out  dd  stdout
envar   db  'PINHOLE='  ; Exactly 8 bytes, or 2 dwords long
pinhole db  '04,',      ; Bender's constant (0.04)
connors db  '037', 0Ah  ; Connors' constant
usg db  'Usage: pinhole [-b] [-c] [-e] [-p <value>] [-o <outfile>] [-i <infile>]', 0Ah
usglen  equ $-usg
iemsg   db  "pinhole: Can't open input file", 0Ah
iemlen  equ $-iemsg
oemsg   db  "pinhole: Can't create output file", 0Ah
oemlen  equ $-oemsg
pinmsg  db  "pinhole: The PINHOLE constant must not be 0", 0Ah
pinlen  equ $-pinmsg
toobig  db  "pinhole: The PINHOLE constant may not exceed 18 decimal places", 0Ah
biglen  equ $-toobig
huhmsg  db  9, '???'
separ   db  9, '???'
sep2    db  9, '???'
sep3    db  9, '???'
sep4    db  9, '???', 0Ah
huhlen  equ $-huhmsg
header  db  'focal length in millimeters,pinhole diameter in microns,'
    db  'F-number,normalized F-number,F-5.6 multiplier,stops '
    db  'from F-5.6', 0Ah
headlen equ $-header

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE
dbuffer resb    20      ; decimal input buffer
bbuffer resb    10      ; BCD buffer

section .text
align 4
huh:
    call    write
```

```
    push    dword huhlen
    push    dword huhmsg
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    ret

align 4
perr:
    push    dword pinlen
    push    dword pinmsg
    push    dword stderr
    sys.write
    push    dword 4     ; return failure
    sys.exit

align 4
consttoobig:
    push    dword biglen
    push    dword toobig
    push    dword stderr
    sys.write
    push    dword 5     ; return failure
    sys.exit

align 4
ierr:
    push    dword iemlen
    push    dword iemsg
    push    dword stderr
    sys.write
    push    dword 1     ; return failure
    sys.exit

align 4
oerr:
    push    dword oemlen
    push    dword oemsg
    push    dword stderr
    sys.write
    push    dword 2
    sys.exit

align 4
usage:
    push    dword usglen
    push    dword usg
    push    dword stderr
    sys.write
    push    dword 3
    sys.exit
```

```
align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]
    sub esi, esi

.arg:
    pop ecx
    or  ecx, ecx
    je  near .getenv        ; no more arguments

    ; ECX contains the pointer to an argument
    cmp byte [ecx], '-'
    jne usage

    inc ecx
    mov ax, [ecx]
    inc ecx

.o:
    cmp al, 'o'
    jne .i

    ; Make sure we are not asked for the output file twice
    cmp dword [fd.out], stdout
    jne usage

    ; Find the path to output file - it is either at [ECX+1],
    ; i.e., -ofile --
    ; or in the next argument,
    ; i.e., -o file

    or  ah, ah
    jne .openoutput
    pop ecx
    jecxz   usage

.openoutput:
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
    jc  near oerr

    add esp, byte 12
    mov [fd.out], eax
    jmp short .arg

.i:
```

```
    cmp al, 'i'
    jne .p

    ; Make sure we are not asked twice
    cmp dword [fd.in], stdin
    jne near usage

    ; Find the path to the input file
    or  ah, ah
    jne .openinput
    pop ecx
    or  ecx, ecx
    je near usage

.openinput:
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  near ierr       ; open failed

    add esp, byte 8
    mov [fd.in], eax
    jmp .arg

.p:
    cmp al, 'p'
    jne .c
    or  ah, ah
    jne .pcheck

    pop ecx
    or  ecx, ecx
    je  near usage

    mov ah, [ecx]

.pcheck:
    cmp ah, '0'
    jl  near usage
    cmp ah, '9'
    ja  near usage
    mov esi, ecx
    jmp .arg

.c:
    cmp al, 'c'
    jne .b
    or  ah, ah
    jne near usage
    mov esi, connors
    jmp .arg
```

```
.b:
    cmp al, 'b'
    jne .e
    or  ah, ah
    jne near usage
    mov esi, pinhole
    jmp .arg

.e:
    cmp al, 'e'
    jne near usage
    or  ah, ah
    jne near usage
    mov al, ','
    mov [huhmsg], al
    mov [separ], al
    mov [sep2], al
    mov [sep3], al
    mov [sep4], al
    jmp .arg

align 4
.getenv:
    ; If ESI = 0, we did not have a -p argument,
    ; and need to check the environment for "PINHOLE="
    or  esi, esi
    jne .init

    sub ecx, ecx

.nextenv:
    pop esi
    or  esi, esi
    je  .default    ; no PINHOLE envar found

    ; check if this envar starts with 'PINHOLE='
    mov edi, envar
    mov cl, 2       ; 'PINHOLE=' is 2 dwords long
rep cmpsd
    jne .nextenv

    ; Check if it is followed by a digit
    mov al, [esi]
    cmp al, '0'
    jl  .default
    cmp al, '9'
    jbe .init
    ; fall through

align 4
```

```
.default:
    ; We got here because we had no -p argument,
    ; and did not find the PINHOLE envar.
    mov esi, pinhole
    ; fall through

align 4
.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    sub edx, edx
    mov edi, dbuffer+1
    mov byte [dbuffer], '0'

    ; Convert the pinhole constant to real
.constloop:
    lodsb
    cmp al, '9'
    ja  .setconst
    cmp al, '0'
    je  .processconst
    jb  .setconst

    inc dl

.processconst:
    inc cl
    cmp cl, 18
    ja  near consttoobig
    stosb
    jmp short .constloop

align 4
.setconst:
    or  dl, dl
    je  near perr

    finit
    fild    dword [tthou]

    fld1
    fild    dword [ten]
    fdivp   st1, st0

    fild    dword [thousand]
    mov edi, obuffer

    mov ebp, ecx
    call    bcdload
```

```
.constdiv:
    fmul    st0, st2
    loop    .constdiv

    fld1
    fadd    st0, st0
    fadd    st0, st0
    fld1
    faddp   st1, st0
    fchs

    ; If we are creating a CSV file,
    ; print header
    cmp byte [separ], ','
    jne .bigloop

    push    dword headlen
    push    dword header
    push    dword [fd.out]
    sys.write

.bigloop:
    call    getchar
    jc  near done

    ; Skip to the end of the line if you got '#'
    cmp al, '#'
    jne .num
    call    skiptoeol
    jmp short .bigloop

.num:
    ; See if you got a number
    cmp al, '0'
    jl  .bigloop
    cmp al, '9'
    ja  .bigloop

    ; Yes, we have a number
    sub ebp, ebp
    sub edx, edx

.number:
    cmp al, '0'
    je  .number0
    mov dl, 1

.number0:
    or  dl, dl      ; Skip leading 0's
    je  .nextnumber
    push    eax
```

```
    call    putchar
    pop eax
    inc ebp
    cmp ebp, 19
    jae .nextnumber
    mov [dbuffer+ebp], al

.nextnumber:
    call    getchar
    jc  .work
    cmp al, '#'
    je  .ungetc
    cmp al, '0'
    jl  .work
    cmp al, '9'
    ja  .work
    jmp short .number

.ungetc:
    dec esi
    inc ebx

.work:
    ; Now, do all the work
    or  dl, dl
    je  near .work0

    cmp ebp, 19
    jae near .toobig

    call    bcdload

    ; Calculate pinhole diameter

    fld st0 ; save it
    fsqrt
    fmul    st0, st3
    fld st0
    fmul    st5
    sub ebp, ebp

    ; Round off to 4 significant digits
.diameter:
    fcom    st0, st7
    fstsw   ax
    sahf
    jb  .printdiameter
    fmul    st0, st6
    inc ebp
    jmp short .diameter
```

```
.printdiameter:
    call    printnumber ; pinhole diameter

    ; Calculate F-number

    fdivp   st1, st0
    fld st0

    sub ebp, ebp

.fnumber:
    fcom    st0, st6
    fstsw   ax
    sahf
    jb  .printfnumber
    fmul    st0, st5
    inc ebp
    jmp short .fnumber

.printfnumber:
    call    printnumber ; F number

    ; Calculate normalized F-number
    fmul    st0, st0
    fld1
    fld st1
    fyl2x
    frndint
    fld1
    fscale
    fsqrt
    fstp    st1

    sub ebp, ebp
    call    printnumber

    ; Calculate time multiplier from F-5.6

    fscale
    fld st0

    ; Round off to 4 significant digits
.fmul:
    fcom    st0, st6
    fstsw   ax
    sahf

    jb  .printfmul
    inc ebp
    fmul    st0, st5
    jmp short .fmul
```

```
.printfmul:
    call    printnumber ; F multiplier

    ; Calculate F-stops from 5.6

    fld1
    fxch    st1
    fyl2x

    sub ebp, ebp
    call    printnumber

    mov al, 0Ah
    call    putchar
    jmp .bigloop

.work0:
    mov al, '0'
    call    putchar

align 4
.toobig:
    call    huh
    jmp .bigloop

align 4
done:
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close

    finit

    ; return success
    push    dword 0
    sys.exit

align 4
skiptoeol:
    ; Keep reading until you come to cr, lf, or eof
    call    getchar
    jc  done
    cmp al, 0Ah
    jne .cr
    ret
```

```
.cr:
    cmp al, 0Dh
    jne skiptoeol
    ret

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    clc
    ret

read:
    jecxz   .read
    call    write

.read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .empty
    sub eax, eax
    ret

align 4
.empty:
    add esp, byte 4
    stc
    ret

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
```

```
write:
    jecxz   .ret    ; nothing to write
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
.ret:
    ret

align 4
bcdload:
    ; EBP contains the number of chars in dbuffer
    push    ecx
    push    esi
    push    edi

    lea ecx, [ebp+1]
    lea esi, [dbuffer+ebp-1]
    shr ecx, 1

    std

    mov edi, bbuffer
    sub eax, eax
    mov [edi], eax
    mov [edi+4], eax
    mov [edi+2], ax

.loop:
    lodsw
    sub ax, 3030h
    shl al, 4
    or  al, ah
    mov [edi], al
    inc edi
    loop    .loop

    fbld    [bbuffer]

    cld
    pop edi
    pop esi
    pop ecx
    sub eax, eax
    ret

align 4
```

```
printnumber:
    push    ebp
    mov al, [separ]
    call    putchar

    ; Print the integer at the TOS
    mov ebp, bbuffer+9
    fbstp   [bbuffer]

    ; Check the sign
    mov al, [ebp]
    dec ebp
    or  al, al
    jns .leading

    ; We got a negative number (should never happen)
    mov al, '-'
    call    putchar

.leading:
    ; Skip leading zeros
    mov al, [ebp]
    dec ebp
    or  al, al
    jne .first
    cmp ebp, bbuffer
    jae .leading

    ; We are here because the result was 0.
    ; Print '0' and return
    mov al, '0'
    jmp putchar

.first:
    ; We have found the first non-zero.
    ; But it is still packed
    test    al, 0F0h
    jz  .second
    push    eax
    shr al, 4
    add al, '0'
    call    putchar
    pop eax
    and al, 0Fh

.second:
    add al, '0'
    call    putchar

.next:
    cmp ebp, bbuffer
```

```
    jb  .done

    mov al, [ebp]
    push    eax
    shr al, 4
    add al, '0'
    call    putchar
    pop eax
    and al, 0Fh
    add al, '0'
    call    putchar

    dec ebp
    jmp short .next

.done:
    pop ebp
    or  ebp, ebp
    je  .ret

.zeros:
    mov al, '0'
    call    putchar
    dec ebp
    jne .zeros

.ret:
    ret
```

Код следует тому же формату, что и все остальные фильтры, которые мы видели ранее, с одним небольшим исключением:

> Мы больше не предполагаем, что конец ввода означает конец задач, как мы привыкли в фильтрах, *ориентированных на символы*.
>
> Этот фильтр не обрабатывает символы. Он обрабатывает *язык* (хотя и очень простой, состоящий только из чисел).
>
> Когда у нас больше нет входных данных, это может означать одно из двух:
>
> - Мы закончили и можем выйти. Это то же самое, что и раньше.
> - Последний прочитанный символ был цифрой. Мы сохранили его в конце буфера преобразования ASCII в число с плавающей точкой. Теперь нам нужно преобразовать содержимое этого буфера в число и записать последнюю строку нашего вывода.

По этой причине мы изменили наши подпрограммы `getchar` и `read`, чтобы они возвращались с *сброшенным* флагом `carry`, когда получают очередной символ из ввода, или с *установленным* флагом `carry`, когда ввода больше нет.

Конечно, мы по-прежнему используем магию ассемблера для этого! Внимательно посмотрите на `getchar`. Он *всегда* возвращает *очищенный* `флаг переноса`.

Тем не менее, наш основной код использует `флаг переноса` для определения момента завершения — и это работает.

Волшебство кроется в `read`. Каждый раз, когда он получает больше входных данных от системы, он просто возвращается к `getchar`, который извлекает символ из входного буфера, *сбрасывает* флаг переноса (`carry flag`) и возвращает управление.

Но когда `read` больше не получает входных данных от системы, он *не* возвращается к `getchar` вообще. Вместо этого, инструкция `add esp, byte 4` добавляет `4` к `ESP`, *устанавливает* флаг переноса (`carry flag`) и возвращает управление.

Итак, куда же она возвращается? Каждый раз, когда программа использует операцию `call`, микропроцессор делает `push` для адрес возврата, то есть сохраняет его на вершине стека (не стека FPU, а системного стека, который находится в памяти). Когда программа использует операцию `ret`, микропроцессор делает `pop` для значения возврата из стека и переходит по адресу, который там был сохранён.

Но поскольку мы добавили `4` к `ESP` (который является регистром указателя стека), мы фактически вызвали у микропроцессора лёгкий случай *амнезии*: он больше не помнит, что именно `getchar вызвал read`.

И поскольку `getchar` не делал `push` ни для чего перед вызовом `read`, верхушка стека теперь содержит адрес возврата к тому, что или кто вызывал `getchar`. С точки зрения этого вызывающего, он вызывал `getchar`, который вызвал `ret` с установленным `флагом переноса`!

Помимо этого, процедура `bcdload` оказывается втянута в лилипутский конфликт между Биг-Эндианцами и Литл-Эндианцами.

Он преобразует текстовое представление числа в само число: текст хранится в порядке big-

endian, но *упакованный десятичный* формат имеет порядок little-endian.

Для разрешения конфликта мы используем инструкцию процессора `std` в самом начале. Позже мы отменяем его с помощью `cld`: очень важно не вызывать ничего, что может зависеть от стандартного значения *флага направления*, пока активен `std`.

Всё остальное в этом коде должно быть достаточно понятным, при условии, что вы прочитали всю предшествующую главу.

Это классический пример поговорки о том, что программирование требует много размышлений и лишь немного кодирования. Как только мы продумаем каждую мельчайшую деталь, код практически напишется сам.

## 13.6. Использование программы pinhole

Поскольку мы решили сделать так, чтобы программа *игнорировала* любой ввод, кроме чисел (и даже их внутри комментария), мы можем выполнять *текстовые запросы*. Мы не *обязаны* этого делать, но *можем*.

По моему скромному мнению, формирование текстового запроса вместо необходимости следовать очень строгому синтаксису делает программное обеспечение гораздо более дружелюбным к пользователю.

Предположим, мы хотим построить камеру-обскуру для использования плёнки размером 4x5 дюймов. Стандартное фокусное расстояние для такой плёнки составляет около 150 мм. Мы хотим *точно настроить* фокусное расстояние, чтобы диаметр отверстия был как можно более круглым числом. Допустим также, что мы хорошо разбираемся в фотоаппаратах, но немного боимся компьютеров. Вместо того чтобы просто вводить кучу цифр, мы хотим *задать* пару вопросов.

Наша сессия может выглядеть так:

```
% pinhole

Computer,

What size pinhole do I need for the focal length of 150?
150 490 306 362 2930    12
Hmmm... How about 160?
160 506 316 362 3125    12
Let's make it 155, please.
155 498 311 362 3027    12
Ah, let's try 157...
157 501 313 362 3066    12
156?
156 500 312 362 3047    12
That's it! Perfect! Thank you very much!
^D
```

Мы выяснили, что при фокусном расстоянии 150 мм диаметр отверстия должен составлять 490 микрон, или 0,49 мм, но если взять почти идентичное фокусное расстояние 156 мм, можно использовать отверстие диаметром ровно половину миллиметра.

## 13.7. Скриптинг

Поскольку мы выбрали символ `#` для обозначения начала комментария, мы можем рассматривать наше программное обеспечение pinhole как *скриптовый язык*.

Вы, вероятно, видели *сценарии* оболочки, которые начинаются с:

```
#! /bin/sh
```

...или...

```
#!/bin/sh
```

...потому что пробел после `#!` необязателен.

Когда UNIX® получает запрос на выполнение исполняемого файла, который начинается с `#!`, он предполагает, что это скрипт. Он добавляет команду к остальной части первой строки скрипта и пытается выполнить её.

Предположим, что мы установили pinhole в /usr/local/bin/, теперь мы можем написать скрипт для расчёта различных диаметров отверстий, подходящих для различных фокусных расстояний, обычно используемых с плёнкой 120.

Скрипт может выглядеть примерно так:

```
#! /usr/local/bin/pinhole -b -i
# Find the best pinhole diameter
# for the 120 film

### Standard
80

### Wide angle
30, 40, 50, 60, 70

### Telephoto
100, 120, 140
```

Поскольку 120 — это плёнка среднего размера, мы можем назвать этот файл `medium`.

Мы можем установить права на выполнение и запустить его, как если бы это была программа:

```
% chmod 755 medium
% ./medium
```

UNIX® интерпретирует последнюю команду следующим образом:

```
% /usr/local/bin/pinhole -b -i ./medium
```

Он выполнит эту команду и отобразит:

```
80  358 224 256 1562    11
30  219 137 128 586 9
40  253 158 181 781 10
50  283 177 181 977 10
60  310 194 181 1172    10
70  335 209 181 1367    10
100 400 250 256 1953    11
120 438 274 256 2344    11
140 473 296 256 2734    11
```

Теперь введем:

```
% ./medium -c
```

UNIX® интерпретирует это следующим образом:

```
% /usr/local/bin/pinhole -b -i ./medium -c
```

Это даёт ему два конфликтующих параметра: `-b` и `-c` (Использовать константу Бендера и использовать константу Коннорса). Мы запрограммировали его так, что более поздние параметры переопределяют ранние — наша программа будет вычислять все, используя константу Коннорса:

```
80  331 242 256 1826    11
30  203 148 128 685 9
40  234 171 181 913 10
50  262 191 181 1141    10
60  287 209 181 1370    10
70  310 226 256 1598    11
100 370 270 256 2283    11
120 405 296 256 2739    11
140 438 320 362 3196    12
```

Мы решаем, что всё же выбираем константу Бендера. Мы хотим сохранить её значения в

виде файла с разделителями-запятыми:

```
% ./medium -b -e > bender
% cat bender
focal length in millimeters,pinhole diameter in microns,F-number,normalized F-
number,F-5.6 multiplier,stops from F-5.6
80,358,224,256,1562,11
30,219,137,128,586,9
40,253,158,181,781,10
50,283,177,181,977,10
60,310,194,181,1172,10
70,335,209,181,1367,10
100,400,250,256,1953,11
120,438,274,256,2344,11
140,473,296,256,2734,11
%
```

# 14. Предостережения

Программисты на ассемблере, которые "выросли" на MS-DOS® и Windows®, часто склонны искать короткие пути. Чтение скан-кодов клавиатуры и запись напрямую в видеопамять — это два классических примера подходов, которые в MS-DOS® не только не порицаются, но и считаются правильными.

Причина? И BIOS ПК, и MS-DOS® печально известны своей медленной работой при выполнении этих операций.

Вас может возникнуть соблазн продолжить подобные практики в среде UNIX®. Например, я видел веб-сайт, который объясняет, как получить доступ к скан-кодам клавиатуры на популярном клоне UNIX®.

Это, как правило, **очень плохая идея** в среде UNIX®! Позвольте объяснить почему.

## 14.1. UNIX® защищен

Прежде всего, это может быть просто невозможно. UNIX® работает в защищённом режиме. Только ядро и драйверы устройств имеют прямой доступ к аппаратному обеспечению. Возможно, какой-то конкретный клон UNIX® позволит вам читать скан-коды клавиатуры, но скорее всего настоящая операционная система UNIX® этого не допустит. И даже если одна версия разрешает это, следующая может запретить, так что ваше тщательно разработанное программное обеспечение может в одночасье устареть.

## 14.2. UNIX® — это работа с абстракциями

Но существует гораздо более важная причина не пытаться обращаться к оборудованию напрямую (если, конечно, вы не пишете драйвер устройства), даже в UNIX®-подобных системах, которые позволяют это делать:

*UNIX® — это работа с абстракциями!*

Существует фундаментальное различие в философии проектирования между MS-DOS® и UNIX®. MS-DOS® разрабатывалась как однопользовательская система. Она работает на компьютере, к которому напрямую подключены клавиатура и монитор. Ввод пользователя практически гарантированно поступает с этой клавиатуры. Вывод вашей программы почти всегда отображается на этом экране.

Это НИКОГДА не гарантируется в UNIX®. Довольно часто пользователь UNIX® перенаправляет ввод и вывод программы с помощью конвейеров и перенаправлений:

```
% program1 | program2 | program3 > file1
```

Если вы написали program2, ваш ввод поступает не с клавиатуры, а из вывода program1. Аналогично, ваш вывод не выводится на экран, а становится вводом для program3, чей вывод, в свою очередь, отправляется в file1.

Но это ещё не все! Даже если вы убедились, что ваш ввод поступает с терминала, а вывод отправляется на терминал, нет гарантии, что терминал является ПК: его видеопамять может находиться не там, где вы ожидаете, а клавиатура может генерировать не PC-совместимые скан-коды. Это может быть Macintosh® или любой другой компьютер.

Вот вы, возможно, покачаете головой: мое программное обеспечение написано на языке ассемблера для ПК, как оно может работать на Macintosh®? Но я не говорил, что ваше программное обеспечение будет работать на Macintosh®, а лишь что его терминалом может быть Macintosh®.

В UNIX® терминал не обязательно должен быть напрямую подключён к компьютеру, на котором работает ваше программное обеспечение — он может находиться даже на другом континенте или, например, на другой планете. Вполне возможно, что пользователь Macintosh® в Австралии подключается к системе UNIX® в Северной Америке (или где-либо ещё) через telnet. Программное обеспечение работает на одном компьютере, а терминал находится на другом: если попытаться считать скан-коды, будут получены неверные данные!

То же самое относится и к любому другому оборудованию: файл, который вы читаете, может находиться на диске, к которому у вас нет прямого доступа. Камера, с которой вы считываете изображения, может находиться на космическом корабле, соединённом с вами через спутники.

Вот почему в UNIX® никогда нельзя делать никаких предположений о том, откуда поступают ваши данные и куда они направляются. Всегда позволяйте системе управлять физическим доступом к оборудованию.



Это предостережения, а не абсолютные правила. Возможны исключения. Например, если текстовый редактор определил, что работает на локальной машине, он может захотеть читать скан-коды напрямую для улучшенного управления. Я упоминаю эти предостережения не для того, чтобы сказать

> вам, что делать или чего не делать, а просто чтобы вы осознавали определённые подводные камни, которые ждут вас, если вы только что перешли с MS-DOS® на UNIX®. Конечно, творческие люди часто нарушают правила, и это нормально, пока они осознают, что нарушают их, и понимают почему.

# 15. Благодарности

Это руководство никогда бы не было создано без помощи многих опытных программистов FreeBSD из Список рассылки FreeBSD, посвящённый техническим обсуждениям, которые терпеливо отвечали на мои вопросы и направляли меня в моих попытках изучить внутренние механизмы программирования в системе UNIX® в целом и в FreeBSD в частности.

Томас М. Соммерс открыл дверь для меня. Его Как написать "Hello, world" на ассемблере в FreeBSD? веб-страница стала моей первой встречей с примером программирования на ассемблере под FreeBSD.

Джейк Буркхолдер держал дверь открытой, охотно отвечая на все мои вопросы и предоставляя примеры исходного кода на языке ассемблера.

Copyright ® 2000-2001 G. Adam Stanislav. All rights reserved.