МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

А.В. Столяров

# АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

«ЯЗЫК АССЕМБЛЕРА В ОС UNIX»

Часть II

Москва - 2010

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ»

**Кафедра прикладной математики**
А.В. Столяров

# АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

## «ЯЗЫК АССЕМБЛЕРА В ОС UNIX»

Тексты лекций

Часть II

Москва-2010

УДК [004.438Ассемблер:004.451.9UNIX](076.6)
ББК 32.973.26-018.1Ассемблер.я73-2+32.973.26-018.2я73-2
С 81

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты: д-р техн. наук, проф. В.Л. Кузнецов;
канд. физ-мат. наук, доц. И.Г. Головин

Столяров А.В.

С 81 Архитектура ЭВМ и системное программное обеспечение. Язык ас
семблера в ОС Unix. Часть II. Тексты лекций. – М.: МГТУ ГА, 2010.- с.8
рис. 4.

ISBN 978-5-86311-770-6

В пособии представлено содержание второй части лекций по про
граммированию на языке ассемблера NASM, читаемых в рамках курс
«Архитектура ЭВМ и системное программное обеспечение».

Целью лекций является формирование понимания устройства вычисли
тельных машин и получения опыта работы на уровне машинных команд.

Тексты лекций соответствуют рабочей программе учебной дисципли
ны «Архитектура ЭВМ и системное программное обеспечение» по Учеб
ному плану специальности 230401 для студентов II курса дневного обу
чения.

Рассмотрено и одобрено на заседаниях кафедры 20.04.2010 г. и мето
дического совета 20.04.2010 г.

*Издается в авторской редакции.*

ББК 32.973.26-018.1Ассемблер.я73-2+32.973.26-018.2я73-2
Доп. св. план 2010 г.
поз.63
СТОЛЯРОВ Андрей Викторович
АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.
ЯЗЫК АССЕМБЛЕРА В ОС UNIX
Часть II
Тексты лекций

| | | |
|---|---|---|
| | Подписано в печать 05.07.10 г. | |
| Печать офсетная | Формат 60x84/16 | 4.86 уч.-изд. л. |
| 5.11 усл.печ.л. | Заказ № 1125/830 | Тираж 150 экз. |

*Московский государственный технический университет ГА*
125993 Москва, Кронштадтский бульвар, д. 20
*Редакционно-издательский отдел*
125493 Москва, ул. Пулковская, д.6а

ISBN 978-5-86311-770-6

## ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Настоящее учебное пособие Андрея Викторовича Столярова, называемое далее ≪Произведением≫, защищено действующим авторско-правовым законодательством. Все права на Произведение, предусмотренные действующим законодательством, как имущественные, так и неимущественные, принадлежат его автору. Согласно лицензионному договору между автором и МГТУГА от 24 мая 2010 г., МГТУГА предоставлена простая (неисключительная) лицензия на распространение данного файла в его оригинальной форме (без внесения каких-либо изменений) исключительно в целях обеспечения учебного процесса; договор не предоставляет МГТУГА права на коммерческое использование Произведения.

В дополнение к этому, автор и правообладатель предоставляет неограниченному кругу лиц право использования электронной версии Произведения в порядке и на условиях, изложенных ниже, при условии безоговорочного принятия этими лицами всех условий настоящей Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является нарушением действующего законодательства и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями исходного файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая и изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через вебсайты, оказывающие платные услуги, через сайты коммерческих компаний, а также через сайты, содержащие рекламу любого рода; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, запись данного файла на носители, принадлежащие другим лицам, распространение данного файла через бесплатные файлообменные сети и т.п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

**А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.**

# Предисловие ко второй части

В первой части этого пособия мы рассмотрели архитектуру процессоров линейки i386 и основные машинные команды, доступные в ограниченном режиме работы центрального процессора. При этом мы использовали ассемблер NASM, ограничиваясь лишь общими замечаниями и изредка отвлекаясь, чтобы описать некоторые его возможности, без которых не могли обойтись. Так, в § 1.4 было дано ровно столько пояснений, чтобы можно было понять одну простейшую программу. В § 2.2 нам потребовалось использовать память для хранения данных, и пришлось посвятить § 2.2.1 директивам резервирования памяти и меткам. Прежде чем привести в § 2.6.9 пример сложной подпрограммы, мы вынуждены были в § 2.6.8 рассказать про локальные метки.

Точно так же мы лишь кратко упомянули особенности работы, обусловленные природой мультизадачных операционных систем; мы использовали макросы, не зная, как они устроены; мы вызывали системный компоновщик, не понимая, что он делает и для чего нужен. Теперь, когда мы приблизительно представляем себе, как выглядит программирование на языке ассемблера, пришла пора заполнить все эти пробелы, чему и будет посвящена вторая часть пособия.

Главу 3 мы целиком посвятим изучению ассемблера NASM, начав с более формального, чем раньше, описания синтаксиса его языка. После этого мы изучим возможности его макропроцессора, научившись, кроме всего прочего, самостоятельно создавать макросы, и закончим это всё кратким описанием ключей командной строки, используемых при запуске NASM. В следующей главе мы рассмотрим взаимодействие нашей программы с операционной системой, для чего придётся изучить теоретические основы функционирования операционных систем; мы узнаем, как выполнять системные вызовы в ОС Linux и FreeBSD, что позволит в наших программах отказаться от использования файла `stud_io.inc`. Наконец, последняя глава будет посвящена модульному программированию и принципам работы редактора связей.

# Глава 3. Ассемблер NASM

## § 3.1. Синтаксис языка ассемблера NASM

Основной синтаксической единицей практически любого языка ассемблера (и NASM тут не исключение) является *строка текста*. Этим языки ассемблера отличаются от большинства (хотя и далеко не всех) языков высокого уровня, в которых символ перевода строки приравнивается к обычному пробелу.

Если по каким-либо причинам нам не хватило длины строки, чтобы уместить всё, что мы хотели в ней уместить, то можно воспользоваться средством «склеивания» строк. Если последним символом строки поставить «обратный слэш» (символ «\»), то ассемблер будет считать следующую строку продолжением предыдущей. Отметим, что это гораздо лучше, чем допускать в тексте программы очень длинные строки; обычно строка программы (любой, не только на языке ассемблера) не должна превышать 75 символов, хотя компиляторы этого от нас и не требуют.

Строка текста[1] на языке ассемблера NASM состоит (в общем случае) из четырёх полей: метки, команды, операндов и комментария, причём метка, команда и комментарий являются полями необязательными, что касается операндов, то требования к ним налагаются командой; если команда отсутствует, то отсутствуют и операнды. Могут отсутствовать и все четыре поля; тогда строка оказывается пустой. Ассемблер пустые строки игнорирует, но мы можем использовать их, чтобы визуально разделять между собой части программы.

В качестве метки можно использовать слово, состоящее из латинских букв, цифр, а также символов `'_'`, `'$'`, `'#'`, `'@'`, `'~'`, `''`. и `'?'`,

[1] Здесь и далее под «строкой» понимается в том числе и «логическая» строка, склеенная из нескольких строк с помощью обратных слэшей; в дальнейшем мы не будем уточнять, что имеем в виду именно такие строки.

причём начинаться метка может только с буквы или символов `'_'`, `'?'` и `'.'`; как мы видели в §2.6.8, метки, начинающиеся с точки, считаются *локальными*. Кроме того, в некоторых случаях имя метки можно предварить символом `'$'`; обычно это используется, если нужно создать метку, имя которой совпадает с именем регистра, команды или директивы[2]. Надо отметить, что ассемблер различает регистр букв в именах меток, то есть, например, `'label'`, `'LABEL'`, `'Label'` и `'LaBeL'` — это четыре разные метки.

После метки, если она в строке присутствует, можно поставить символ двоеточия, но не обязательно. Как уже отмечалось, обычно программисты ставят двоеточия после меток, на которые можно передавать управление, и не ставят двоеточия после меток, обозначающих области памяти. Хотя ассемблер и не требует поступать именно так, программа при использовании этого соглашения становится яснее.

В поле команды, если оно присутствует, может быть обозначение машинной команды (возможно, с префиксом `rep`, см. §2.7; существуют и другие префиксы), либо псевдокоманды — директивы специального вида (некоторые из них мы уже рассматривали, и к этому вопросу ещё вернёмся), либо, наконец, ***имя макроса*** (с такими мы тоже встречались, к ним относится, например, использовавшийся в примерах `PRINT`; созданию макросов будет посвящён отдельный параграф). В отличие от меток, в именах машинных команд и псевдокоманд ассемблер регистры букв не различает, так что мы можем с равным успехом написать, например, `mov`, `MOV`, `Mov` и даже `mOv`, хотя так писать, конечно же, не стоит. В именах макросов, как и в именах меток, регистр различается.

Требования к содержимому поля операндов зависят от того, какая конкретно команда, псевдокоманда или макрос указаны в поле команды. Если операндов больше одного, то они разделяются запятой. Отметим, что в поле операндов часто приходится использовать названия регистров, и в этих названиях регистр букв не различается, как и в именах машинных команд.

Читателю, запутавшемуся в том, где же регистр важен, а где нет, можно порекомендовать одно простое правило: **ассемблер `nasm` не различает заглавные и строчные буквы во всех словах, которые он ввёл сам: в именах команд, названиях регистров, директивах, псевдокомандах, обозначениях длины операндов и**

[2]Такое может понадобиться только в случае, если ваша программа состоит из модулей, написанных на разных языках программирования; тогда в других модулях вполне могут встретиться метки, совпадающие по имени с ключевыми словами ассемблера, и может потребоваться возможность на них ссылаться

**типа переходов (слова `byte`, `dword`, `near` и т. п.), но при этом считает заглавные и строчные разными буквами в тех именах, которые вводит пользователь (программист, пишущий на языке ассемблера) — в метках и именах макросов**.

Отметим ещё одно свойство NASM, связанное с записью операндов. **Операнд типа «память» *всегда* записывается с использованием квадратных скобок**. Отметим, что для некоторых других ассемблеров это не так, что порождает постоянную путаницу.

Комментарий обозначается символом «точка с запятой» («`;`»). Начиная с этого символа, весь текст до конца строки ассемблер не принимает во внимание, что позволяет написать там всё, что угодно. Обычно это используют для вставки в текст программы пояснений, предназначенных для тех, кому придётся этот текст прочитать.

## § 3.2. Псевдокоманды

Под ***псевдокомандами*** понимается ряд вводимых ассемблером NASM слов, которые могут использоваться синтаксически так же, как и мнемоники машинных команд, хотя машинными командами на самом деле не являются.

Некоторые такие псевдокоманды, а именно `db`, `dw`, `dd`, `resb`, `resw` и `resd` нам уже известны из § 2.2.1. Отметим только, что кроме перечисленных, NASM поддерживает также псевдокоманды `resq`, `rest`, `dq` и `dt`. Буква `q` в их названиях означает «quadro» — «учетверённое слово» (8 байт), буква `t` — от слова «ten» и означает десятибайтные элементы. Эти псевдокоманды могут потребоваться только в программе, работающей с числами с плавающей точкой (попросту говоря, дробными числами); более того, `dq` и `dt` в качестве инициализаторов допускают только, и исключительно, числа с плавающей точкой (например, `71.361775`). Кроме псевдокоманд `dq` и `dt`, числа с плавающей точкой принимает и псевдокоманда `dd`; это обусловлено тем, что стандарт IEEE 754 предусматривает три формата чисел с плавающей точкой — обычные, двойной точности и повышенной точности, занимающие, соответственно, 4 байта, 8 байт и 10 байт.

Отдельного разговора заслуживает псевдокоманда `equ`, предназначенная для *определения констант*. Эта псевдокоманда всегда применяется в сочетании с меткой, то есть не поставить перед ней метку считается ошибкой. Функция её в том, чтобы связать метку с *явно заданным числом*. Самый простой пример:

```
four    equ 4
```

Здесь мы определили метку `four`, **задающую число 4**. Таким образом, написать теперь, например,

```
        mov eax, four
```

есть то же самое, что и

```
        mov eax, 4
```

Уместно напомнить, что, вообще говоря, *любая метка представляет собой не более чем число*, но метки, введённые другим способом (помечающие другие строки) связываются с ***адресами в памяти*** (которые, разумеется, есть тоже ни что иное как просто числа).

Одно из самых частых применений директивы `equ` — это связать с некоторым именем (меткой) длину массива, только что заданного с помощью директивы `db`, `dw` или любой другой. Для этого используется ***псевдометка*** `$`, которая в каждой строчке, где она появляется, обозначает *текущий адрес*[3]. Например, можно написать так:

```
msg     db "Hello and welcome", 10, 0
msglen  equ $-msg
```

Выражение `$-msg`, представляющее собой разность двух чисел, известных ассемблеру во время его работы, будет просто вычислено прямо во время ассемблирования. Поскольку `$` означает адрес, ставший текущим уже *после* описания строки, а `msg` — адрес *начала* строки, то их разность в точности равна длине строки (в нашем примере 19). К вычислению выражений во время ассемблирования мы вернёмся в § 3.4.

Директива `times` позволяет повторить какую-нибудь команду (или псевдокоманду) заданное количество раз. Например,

```
stars  times 4096 db '*'
```

задаёт область памяти размером в 4096 байт, заполненную кодом символа `'*'`, точно так же, как это сделали бы 4096 одинаковых строк, содержащих директиву `db '*'`.

Иногда может оказаться полезной и псевдокоманда `incbin`, позволяющая создать область памяти, заполненную данными из некоторого внешнего файла. Подробно мы её рассматривать не будем; заинтересованный читатель может изучить эту директиву самостоятельно, обратившись к документации.

[3] Точнее говоря, текущее смещение относительно начала секции

## § 3.3. Константы

***Константы*** в языке ассемблера NASM делятся на четыре категории: целые числа, символьные константы, строковые константы и числа с плавающей точкой.

Как уже говорилось (см. § 2.2.1), ***целочисленные константы*** можно задавать в десятичной, двоичной, шестнадцатеричной и восьмеричной системах счисления. Если просто написать число, состоящее из цифр (и, возможно, знака «минус» в качестве первого символа), то это число будет воспринято ассемблером как десятичное. Шестнадцатеричное число можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$`, как в Borland Pascal (например, `$2af3`), либо поставив, опять таки, перед числом символы `0x`, как в языке Си (`0x2af3`). При использовании символа `$` необходимо следить, чтобы сразу после `$` стояла цифра, а не буква, так что если число начинается с буквы, необходимо добавить `0` (например, `$0f9` вместо просто `$f9`). Это необходимо, чтобы ассемблер не путал запись числа с записью пользовательской метки, перед которыми, как мы уже говорили, иногда тоже ставится знак `$`. Восьмеричное число обозначается добавлением после числа буквы `o` или `q` (например, `634o`, `754q`). Наконец, двоичное число обозначается буквой `b` (`10011011b`).

***Символьные константы*** и ***строковые константы*** очень похожи друг на друга, и, более того, в любом месте, где по смыслу должна быть строковая константа, можно употребить и символьную. Разница между строковыми и символьными константами заключается только в их длине: под *символьной* константой подразумевается такая константа, которая укладывается в длину «двойного слова» (то есть содержит не более 4 символов) и может, в силу этого, рассматриваться как альтернативная запись целого числа (либо битовой строки).

И символьные, и строковые константы могут записываться как с помощью двойных кавычек, так и с помощью апострофов. Это позволяет использовать в строках и сами символы апострофов и кавычек: если строка содержит символ кавычек одного типа, то её заключают в кавычки другого типа (см. пример в ч. 1 на стр. 39).

Символьные константы, содержащие меньше 4 символов, считаются синонимами целых чисел, младшие байты которых равны кодам символов из константы, а недостающие старшие байты заполнены нулями. При использовании символьных констант следует помнить, что целые числа в компьютерах с процессорами i386 записываются в обратном порядке байтов, то есть младший байт идёт первым. В то же время, по смыслу строки (и символьной константы) код первой буквы должен в памяти размещаться первым. Поэтому, например, кон-

станта 'abcd' эквивалентна числу 64636261h: 64h — это код буквы d, 61h — код буквы a, и в обоих случаях байт со значением 61h стоит первым, а 64h — последним.

В некоторых случаях ассемблер воспринимает в качестве строковых и такие константы, которые достаточно коротки, чтобы считаться и символьными. Это происходит, например, если ассемблер видит символьную константу длиной более 1 символа в параметрах директивы db или константу длиной более двух символов в параметрах директивы dw.

***Константы с плавающей точкой***, задающие дробные числа, синтаксически отличаются от целочисленных констант наличием десятичной точки. Учтите, что **целочисленная константа 1 и константа 1.0 не имеют между собой ничего общего!** Для наглядности отметим, что битовая запись числа с плавающей точкой 1.0 одинарной точности (то есть запись, занимающая 4 байта, так же, как и для целого числа) эквивалентна записи целого числа 3f800000h (1065353216 в десятичной записи). Константу с плавающей точкой можно задать и в «экспоненциальном» виде, используя букву e или E. Например, 1.0e-5 есть то же самое, что и 0.00001. Обратите внимание, что десятичная точка по-прежнему обязательна.

# § 3.4. Вычисление выражений во время ассемблирования

Ассемблер NASM в некоторых случаях вычисляет встретившиеся ему арифметические выражения непосредственно во время ассемблирования. Важно понимать, что **в итоговый машинный код попадают только вычисленные результаты, а не сами действия по их вычислению**. Естественно, для вычисления выражения во время ассемблирования необходимо, чтобы такое выражение не содержало никаких неизвестных: всё, что нужно для вычисления, должно быть известно ассемблеру во время его работы.

## § 3.4.1. Вычисляемые выражения и операции в них

Выражение, вычисляемое ассемблером, должно быть **целочисленным**, то есть состоять из целочисленных констант и меток, и использовать операции из следующего списка:

- + и - — сложение и вычитание
- * — умножение;

- `/` и `%` — целочисленное деление и остаток от деления (для беззнаковых целых чисел);
- `//` и `%%` — целочисленное деление и остаток от деления (для знаковых целых чисел);
- `&`, `|`, `^` — операции побитового «и», «или», «исключающего или»;
- `<<` и `>>` — операции побитового сдвига влево и вправо;
- унарные операции `-` и `+` используются в их обычной роли: `-` меняет знак числа на противоположный, `+` не делает ничего;
- унарная операция `~` обозначает побитовое отрицание.

При применении операций `%` и `%%` необходимо обязательно оставлять пробельный символ после знака операции, чтобы ассемблер не перепутал их с *макродирективами* (макродирективы мы рассмотрим позже).

Ещё одна унарная операция, `seg`, для нас неприменима ввиду отсутствия сегментов в «плоской» модели памяти.

Унарные операции имеют самый высокий приоритет, следом за ними идут операции умножения, деления и остатка от деления, ещё ниже приоритет у операций сложения и вычитания. Далее (в порядке убывания приоритета) идут операции сдвигов, операция `&`, затем операция `^`, и замыкает список операция `|`, имеющая самый низкий приоритет. Порядок выполнения операций можно изменить, применив круглые скобки.

## § 3.4.2. Критические выражения

Ассемблер анализирует исходный текст в два прохода. На первом проходе вычисляется размер всех машинных команд и других данных, подлежащих размещению в памяти программы; в результате этого ассемблер устанавливает, какое *числовое значение* должно быть приписано каждой из встретившихся в тексте программы меток. На втором проходе генерируется собственно машинный код и прочее содержимое памяти. Второй проход нужен, чтобы, например, можно было ссылаться на метку, стоящую в тексте *позже*, чем ссылка на неё: когда ассемблер видит метку, скажем, в команде `jmp`, раньше, чем встретится собственно команда, помеченная этой меткой, на первом проходе он не может сгенерировать код, поскольку не знает численного значения метки. На втором проходе все значения уже известны, и никаких проблем с генерированием кода не возникает.

Всё это имеет прямое отношение к механизму вычисления выражений. Ясно, что выражение, содержащее метку, ассемблер может вычислить на первом проходе только в случае, если метка стояла в тексте раньше, чем вычисляемое выражение; в противном случае вычисление выражения приходится отложить до второго прохода.

Ничего страшного в этом нет, **если только значение выражения не влияет на размер команды, выделяемой области памяти и т. п.**, то есть от значения этого выражения не зависят численные значения, которые нужно будет приписать дальнейшим встреченным меткам. Если же это условие не выполнено, то невозможность вычислить выражение на первом проходе приведёт к невозможности выполнить задачу первого прохода — определить численные значения всех меток. Более того, в некоторых случаях не помогло бы никакое количество проходов, даже если бы ассемблер это умел. В документации к ассемблеру NASM приведён такой пример:

```
          times (label-$) db 0
label:    db      'Where am I?'
```

Здесь строчка с директивой `times` должна ввести столько нулевых байтов, насколько метка `label` отстоит от самой этой строчки — но ведь метка `label` как раз и отстоит от этой строчки настолько, сколько нулевых байтов будет введено. Так сколько же их должно быть введено?!

В связи с этим мы вводим понятие ***критического выражения***: это такое выражение, вычисляемое во время ассемблирования, которое ассемблеру необходимо вычислить во время первого прохода. Критическими ассемблер считает любые выражения, от которых тем или иным образом зависит размер чего бы то ни было, располагаемого в памяти (и которые, следовательно, могут повлиять на значения меток, вводимых позже).

В критических выражениях можно использовать только числовые константы, а также метки, определённые *выше по тексту программы*, чем рассматриваемое выражение. Это гарантирует возможность вычисления выражения на первом проходе.

Кроме аргумента директивы `times`, к категории критических относятся, например, выражения в аргументах псевдокоманд `resb`, `resw` и др., а также в некоторых случаях — выражения в составе исполнительных адресов, которые могут повлиять на итоговый размер ассемблируемой команды. Так, команды «`mov eax,[ebx]`», «`mov eax,[ebx+10]`» и «`mov eax,[ebx+10000]`» дают в результате, соответственно, 2 байта, 3 байта и 6 байтов кода, поскольку число, входящее в состав исполни-

тельного адреса, в первом случае занимает всего 1 байт, во втором — 2, а в последнем — 4; но сколько памяти займёт команда

```
mov eax, [ebx+label]
```

если значение `label` пока не определено? Впрочем, этих трудностей можно избежать, если внутри исполнительного адреса в явном виде указать разрядность словом `byte`, `word` или `dword`. Так, если написать

```
mov eax, [dword ebx + label]
```

то, даже если значение `label` ещё не известно, длина его (и, как следствие, длина всей машинной команды) уже указана.

### § 3.4.3. Выражения в составе исполнительного адреса

На рис. 2.2 (см. § 2.2) мы приводили общий вид исполнительного адреса (операндов типа «память») с точки зрения машинных команд. Ассемблер NASM может воспринимать и более сложные выражения в квадратных скобках, лишь бы их было возможно привести к указанному виду. Так, например, в команде

```
mov eax, [5*ebx]
```

используется умножение на число 5, что вроде бы запрещено (умножать можно только на 1, 2, 4 и 8), но ассемблер справляется с этой сложностью, приведя в команде операнд к виду `[ebx+4*ebx]`, который уже вполне корректен. Если же рассмотреть команду

```
mov eax, [ebx+4*ecx+5*x+y]
```

в которой `x` и `y` — некоторые метки, то и с этим ассемблер справится, попросту *вычислив* выражение `5*x+y` и получив в итоге одно число, что уже вполне соответствует общему виду исполнительного адреса.

Необходимо только помнить, что, если только в явном виде не указать нужную разрядность, такие выражения будут считаться *критическими*, то есть должны зависеть только от меток, уже введённых к моменту рассмотрения выражения (см. предыдущий параграф).

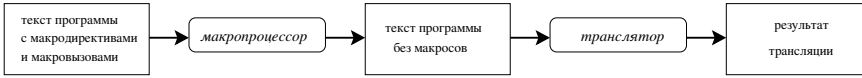


Рис. 3.1. Схема работы макропроцессора

# § 3.5. Макросредства и макропроцессор

## § 3.5.1. Основные понятия

Безотносительно языка программирования, под ***макропроцессором*** понимают программное средство, которое получает на вход некоторый текст и, пользуясь указаниями, данными в самом тексте, частично преобразует его, давая на выходе, в свою очередь, текст, но уже не имеющий указаний к преобразованию.

В применении к языкам программирования макропроцессор — это преобразователь исходного текста программы, обычно совмещённый с компилятором; результатом работы макропроцессора является **текст на языке программирования**, который потом уже обрабатывается компилятором в соответствии с правилами языка (см. рис. 3.1).

Поскольку языки ассемблера обычно весьма бедны по своим изобразительным возможностям (если сравнивать их с языками высокого уровня), то, чтобы хоть как-то компенсировать программистам неудобства, обычно ассемблеры снабжают очень мощными макропроцессорами. В частности, рассматриваемый нами ассемблер NASM содержит в себе алгоритмически полный макропроцессор, который мы можем при желании заставить написать за нас едва ли не всю программу.

С макросами мы уже встречались: так, часто использовавшиеся нами `PRINT` и `FINISH` представляют собой именно макросы, или, точнее, ***имена макросов***.

Вообще, ***макросом*** называют некоторое правило, в соответствии с которым фрагмент программы, содержащий определённое слово, должен быть преобразован. Само это слово называют ***именем макроса***; часто вместо термина «имя макроса» используют просто слово «макрос», хотя это и не совсем верно.

Макрос и его имя необходимо определить, то есть, во-первых, указать макропроцессору, что некий идентификатор отныне считается именем макроса (так что его появление в тексте программы требует вмешательства макропроцессора), и, во-вторых, задать то правило, по кото-

рому макропроцессор должен действовать, встретив это имя. Фрагмент программы, определяющий макрос, называют ***макроопределением***.

Когда макропроцессор встречает в тексте программы имя макроса и параметры (так называемый ***вызов макроса***, или ***макровызов***), он *заменяет* имя макроса (и, возможно, параметры, относящиеся к нему) неким фрагментом текста, полученным в соответствии с определением макроса. Такая замена называется ***макроподстановкой*** или (реже) *макрорасширением*[4].

Бывает и так, что макропроцессор производит преобразование текста программы, не видя ни одного имени макроса, но повинуясь ещё более прямым указаниям, выраженным в виде ***макродирективов***. Одну такую макродирективу мы уже знаем: это директива `%include`, которая приказывает макропроцессору заменить её саму на содержимое файла, указанного параметром директивы. Так, привычная нам строка

```
%include "stud_io.inc"
```

заменяется на всё, что есть в файле `"stud_io.inc"`.

## § 3.5.2. Простейшие примеры макросов

Чтобы составить представление о том, как можно воспользоваться макропроцессором и для чего он нужен, приведём два простых примера.

Как мы видели из §§ 2.6.6, 2.6.7 и 2.6.9, запись вызова подпрограммы на языке ассемблера занимает несколько строк (если быть точными, $2+n$, где $n$ — число параметров подпрограммы). Это не всегда удобно, особенно для людей, привыкших к языкам высокого уровня.

Пользуясь механизмом макросов, мы можем изрядно сократить запись вызова подпрограммы. Для этого мы опишем макросы `pcall1`, `pcall2` и т. д., для вызова, соответственно, процедуры от одного параметра, двух параметров и т. д. С помощью таких макросов запись вызова процедуры сократится до одной строчки; например, вместо

```
push edx
push dword mylabel
push dword 517
call myproc
add esp, 12
```

можно будет написать

[4] Термин ≪макрорасширение≫ — это не слишком удачная калька с соответствующего английского термина ≪macro expansion≫

```
pcall3 myproc, dword 517, dword mylabel, edx
```

что, конечно, гораздо удобнее и понятнее. Позже, разобравшись с макроопределениями глубже, мы перепишем эти макросы, вместо них введя один макрос `pcall`, работающий для любого количества аргументов, но пока для примера ограничимся частными случаями.

Итак, пишем макроопределение:

```
%macro pcall1 2      ; 2 -- кол-во параметров макроса
        push %2
        call %1
        add esp, 4
%endmacro
```

Мы описали ***многострочный макрос*** с именем `pcall1`, принимающий два параметра: имя вызываемой процедуры для команды `call` и аргумент процедуры для занесения в стек. Строки, написанные между директивами `%macro` и `%endmacro`, составляют ***тело макроса*** — шаблон для текста, который должен получиться в результате макроподстановки. Сама макроподстановка в данном случае будет довольно простой: макропроцессор только заменит вхождения `%1` и `%2` соответственно на первый и второй параметры, заданные в макровызове.

Если после такого определения в тексте нашей программы встретится строка вида

```
pcall1 proc, eax
```

макропроцессор воспримет эту строку как макровызов и выполнит макроподстановку в соответствии с вышеприведённым макроопределением, считая первым параметром слово `proc`, вторым параметром слово `eax` и подставляя их вместо `%1` и `%2`. В результате получится следующий фрагмент:

```
push eax
call proc
add esp, 4
```

Аналогичным образом опишем макросы `pcall2` и `pcall3`:

```
%macro pcall2 3
        push %3
        push %2
        call %1
```

```
        add esp, 8
%endmacro
%macro pcall3 4
        push %4
        push %3
        push %2
        call %1
        add esp, 12
%endmacro
```

Для полноты можно дописать также и макрос `pcall0`:

```
%macro pcall0 1
        call %1
%endmacro
```

Конечно, такой макрос, в отличие от предыдущих, ничуть не сокращает объём программы, но зато он позволит нам все вызовы подпрограмм оформить единообразно.

Описание макросов `pcall4`, `pcall5` и т. д. до `pcall8` оставляем читателю в качестве упражнения; заодно для самопроверки ответьте на вопрос, почему мы предлагаем остановиться именно на `pcall8`, а не, например, на `pcall9` или `pcall12`.

Рассмотренный нами пример использовал *многострочный макрос*; как мы убедились, вызов многострочного макроса синтаксически выглядит точно так же, как использование машинных команд или псевдокоманд: вместо имени команды пишется имя макроса, затем через запятую перечисляются параметры. При этом многострочный макрос всегда преобразуется в одну или несколько *строк* на языке ассемблера.

Но что если, к примеру, нам нужно сгенерировать с помощью макроса некоторую *часть строки*, а не фрагмент из нескольких строк? Такая потребность тоже возникает довольно регулярно. Так, в примере, приведённом в § 2.6.9, видно, что внутри процедур очень часто приходится использовать конструкции вроде `[ebp+12]`, `[ebp-4]` и т. п. для обращения к параметрам процедуры и её локальным переменным. В принципе, к этим конструкциям несложно привыкнуть; но можно пойти и другим путём, применив ***однострочные макросы***.

Для начала напишем следующие[5] макроопределения:

[5]Здесь и далее в наших примерах мы предполагаем, что все параметры процедур и все локальные переменные всегда представляют собой ≪двойные слова≫, то есть

```
%define arg1 ebp+8
%define arg2 ebp+12
%define arg3 ebp+16
%define local1 ebp-4
%define local2 ebp-8
%define local3 ebp-12
```

В дополнение к ним допишем ещё и такое:

```
%define arg(n) ebp+(4*n)+4
%define local(n) ebp-(4*n)
```

Теперь к параметру процедуры можно обратиться так:

```
mov eax, [arg1]
```

или так (если, например, не хватило описанных макросов)

```
mov [arg(7)], edx
```

В принципе мы могли и квадратные скобки включить внутрь макросов, чтобы не писать их каждый раз. Например, если изменить определение макроса `arg1` на следующее:

```
%define arg1 [ebp+8]
```

то соответствующий макровызов стал бы выглядеть так:

```
mov eax, arg1
```

Мы не сделали этого из соображений сохранения наглядности. Ассемблер NASM поддерживает, как мы уже знаем, соглашение о том, что любое обращение к памяти оформляется с помощью квадратных скобок, если же их нет, то мы имеем дело с непосредственным или регистровым операндом. Программист, привыкший к этому соглашению, при чтении программы будет вынужден прилагать лишние усилия, чтобы вспомнить, что `arg1` в данном случае не метка, а имя макроса, так что здесь происходит именно обращение к памяти, а не загрузка в регистр адреса метки. Понятности программы такие вещи отнюдь не способствуют. Учтите, что и вы сами, будучи даже автором программы, можете за несколько дней начисто забыть, что́ же имелось в виду, и тогда экономия двух символов (скобок) обернётся для вас потерей бесценного времени.

---

имеют размер 4 байта; на самом деле, конечно, это не всегда так, но нам сейчас важнее иллюстративная ценность примера

## § 3.5.3. Однострочные макросы; макропеременные

Как видно из примеров предыдущего параграфа, однострочный макрос — это такой макрос, определение которого состоит из одной строки, а его вызов разворачивается во фрагмент строки текста (то есть может использоваться для генерации *части* строки).

Отметим, что единожды определённый макрос можно при необходимости *переопределить*, просто вставив в текст программы ещё одно определение того же самого макроса. С того момента, как макропроцессор «увидит» новое определение, он будет использовать его вместо старого. Таким образом, одно и то же имя макроса в разных местах программы может означать разные вещи и раскрываться в разные фрагменты текста. Более того, макрос вообще можно убрать, воспользовавшись директивой `%undef`; встретив такую директиву, макропроцессор немедленно «забудет» о существовании макроса.

Представляет интерес вопрос о том, что будет, если в определении одного макроса использовать вызов другого макроса, а этот последний, в свою очередь, время от времени переопределять.

Если для описания однострочного макроса `A` использовать уже знакомую нам директиву `%define` и в её теле использовать макровызов макроса `B`, то этот макровызов в самой директиве не раскрывается; макропроцессор оставляет вхождение макроса `B` как оно есть до тех пор, пока не встретит вызов макроса `A`. Когда же будет выполнена макроподстановка для `A`, в её результате будет содержаться `B`, и для него макропроцессор, в свою очередь, выполнит макроподстановку. Таким образом, будет использовано то определение макроса `B`, которое было актуальным в момент подстановки `A`.

Поясним сказанное на примере. Пусть мы ввели два макроса:

```
%define        thenumber      25
%define        mkvar          dd thenumber
```

Если теперь написать в программе строчку

```
var1           mkvar
```

то макропроцессор «превратит» её в строку

```
var1           dd 25
```

Если теперь переопределить `thenumber` и снова воспользоваться вызовом `mkvar`:

```
%define         thenumber       36
var2            mkvar
```

то результатом работы макропроцессора будет строка, содержащая именно число 36:

```
var2            dd 36
```

несмотря на то, что сам макрос `mkvar` мы не изменяли. Такая стратегия макроподстановок называется «ленивой[6]». Однако ассембер NASM позволяет применять и другую стратегию, называемую *энергичной*, для чего предусмотрена директива `%xdefine`. Эта директива полностью аналогична директиве `%define` с той только разницей, что, если в теле описания макроса встречаются макровызовы, макропроцессор производит их макроподстановки незамедлительно, не дожидаясь, пока пользователь вызовет описываемый макрос. Так, если в вышеприведённом примере заменить директиву `%define` в описании макроса `mkvar` на `%xdefine`:

```
%define         thenumber       25
%xdefine        mkvar           dd thenumber
var1            mkvar
%define         thenumber       36
var2            mkvar
```

то обе получившиеся строки будут содержать число 25:

```
var1            dd 25
var2            dd 25
```

Переопределение макроса `thenumber` теперь не в силах повлиять на работу макроса `mkvar`, поскольку тело макроса `mkvar` на этот раз не содержит слова `thenumber`: обрабатывая определение `mkvar`, макропроцессор подставил вместо слова `thenumber` его значение (25).

Иногда возникает потребность связать с именем макроса не просто строку, а число, являющееся результатом вычисления арифметического выражения. Ассемблер NASM позволяет это сделать, используя директиву `%assign`. В отличие от `%define` и `%xdefine`, эта директива не только выполняет все необходимые подстановки в теле макроопределения,

[6] Такое название является калькой английского *lazy* и частично оправдано тем, что макропроцессор как бы «ленится» выполнять макроподстановку (в данном случае макроса `thenumber`), пока его к этому не вынудят

но и пытается *вычислить* тело как обыкновенное целочисленное арифметическое выражение. Если это не получается, фиксируется ошибка. Так, если написать в программе сначала

```
%assign         var     25
```

а потом

```
%assign         var     var+1
```

то в результате с макроименем `var` будет связано значение 26, которое и будет подставлено, если макропроцессор встретит слово `var` в дальнейшем тексте программы.

Макроимена, вводимые директивой `%assign`, обычно называют ***макропеременными***. Как мы увидим далее, макропеременные являются важным средством, позволяющим задать макропроцессору целую программу, результатом которой может стать очень длинный текст на языке ассемблера.

## § 3.5.4. Условная компиляция

Часто при разработке программ возникает потребность в создании различных версий исполняемого файла с использованием одного и того же исходного текста. Допустим, мы пишем программы на заказ и у нас есть два заказчика Петров и Сидоров, причём программы для них почти одинаковы, но у каждого из двоих имеются специфические потребности, отсутствующие у другого. В такой ситуации хотелось бы, конечно, иметь и поддерживать один исходный текст: в противном случе у нас появятся две копии одного и того же кода, и придётся, например, каждую найденную ошибку исправлять в двух местах. Однако при компиляции версии для Петрова нужно исключить из работы фрагменты, предназначенные для Сидорова, и наоборот.

Подобная потребность возникает и в других ситуациях. Известно, например, что отладочная печать (то есть вставка в программу специальных операций вывода, позволяющих понять, что происходит во время работы программы) является одним из самых универсальных и мощных средств отладки программ; в то же время окончательная версия программы, разумеется, не должна содержать операций отладочной печати, поскольку вся отладочная информация предназначена для программиста, автора программы, а пользователю может только мешать. Проблема в том, что отладка программы — процесс бесконечный, и как

только мы решим, что она завершена и удалим из текста всю отладочную печать, по закону подлости тут же обнаружится очередная ошибка, и нам вновь придётся редактировать наш исходник, чтобы вернуть отладочную печать на место.

Большинство профессиональных компилируемых языков программирования поддерживают для подобных случаев специальные конструкции, называемые ***директивами условной компиляции*** и позволяющие выбирать, какие фрагменты программы компилировать, а какие игнорировать. Обычно отработку директив условной компиляции возлагают на макропроцессор, если, конечно, таковой в языке предусмотрен. Сказанное справедливо, кроме прочего, практически для всех языков ассемблера, включая и наш NASM.

Рассмотрим пример, связанный с отладкой. Допустим, мы написали программу, откомпилировали её и запустили, но она завершается аварийно, и мы не можем понять причину, но думаем, что авария происходит в некоем «подозрительном» фрагменте. Чтобы проверить своё предположение, мы хотим непосредственно перед входом в этот фрагмент и сразу после выхода из него вставить печать соответствующих сообщений. Чтобы нам не пришлось по несколько раз стирать эти сообщения и вставлять их снова, воспользуемся директивами условной компиляции. Выглядеть это будет примерно так:

```
%ifdef DEBUG_PRINT
        PRINT "Entering suspicious section"
        PUTCHAR 10
%endif
;
;     здесь идёт "подозрительная" часть программы
;
%ifdef DEBUG_PRINT
        PRINT "Leaving suspicious section"
        PUTCHAR 10
%endif
```

Здесь `%ifdef` — это одна из ***директив условной компиляции***, означающая «компилировать только в случае, если определён данный однострочный макрос» (в данном случае это макрос `DEBUG_PRINT`). Теперь в начало программы следует вставить строку, определяющую этот символ:

```
%define DEBUG_PRINT
```

Тогда при запуске NASM он «увидит» и откомпилирует фрагменты нашего исходного текста, заключённые между соответствующими `%ifdef` и `%endif`; когда же мы найдём ошибку и отладочная печать будет нам больше не нужна, достаточно будет убрать этот `%define` из начала программы или даже поставить перед ним знак комментария:

```
;%define DEBUG_PRINT
```

и фрагменты, обрамлённые соответствующими директивами, макропроцессор будет попросту игнорировать, так что их можно совершенно спокойно оставить в тексте программы, а не удалять, на случай, если они снова понадобятся.

Забегая вперёд, отметим, что для включения и отключения отладочной печати, оформленной таким образом, можно вообще обойтись без правки исходного текста. Определить макросимвол можно ключом командной строки NASM; в частности, включить отладочную печать из нашего примера можно, вызвав NASM примерно таким образом:

```
nasm -f elf -dDEBUG_PRINT prog.asm
```

что избавляет нас от необходимости вставлять в исходный текст директиву `%define`, а потом её удалять.

Возвращаясь к ситуации с двумя заказчиками, мы можем предусмотреть в программе конструкции, подобные следующей:

```
%ifdef FOR_PETROV
;
; здесь код, предназначенный только для Петрова
;
%elifdef FOR_SIDOROV
;
; а здесь - только для Сидорова
;
%else
; если ни тот символ, ни другой не определены,
; прервём компиляцию и выдадим сообщение об ошибке
%error Please define either FOR_PETROV or FOR_SIDOROV
%endif
```

(директива `%elifdef` — это сокращённая форма записи для `else` и `ifdef`).

При компиляции такой программы нужно будет обязательно указать ключ `-dFOR_PETROV` или `-dFOR_SIDOROV`, иначе NASM начнёт обрабатывать фрагмент, находящийся после `%else`, и, встретив директиву `%error`, выдаст сообщение об ошибке.

Кроме проверки *наличия* макросимвола, можно проверять также и факт *отсутствия* макросимвола (то есть прямо противоположное условие). Это делается директивой `%ifndef` (*if not defined*). Как и для `%ifdef`, для `%ifndef` существует сокращённая запись конструкции с `%else`, она называется `%elifndef`.

Для задания условия, при котором тот или иной фрагмент подлежит или не подлежит компиляции, можно пользоваться не только фактом наличия или отсутствия макроса; NASM поддерживает и другие директивы условной компиляции. Наиболее общей является директива `%if`, в которой условие задаётся арифметико-логическим выражением, вычисляемым во время компиляции. С такими выражениями мы уже встречались в § 3.4.1; для формирования логических выражений набор допустимых операций расширяется операциями `=`, `<`, `>`, `>=`, `<=`, в их обычном смысле, операцию «не равно» можно задать символом `<>`, как в Паскале, или символом `!=`, как в Си; поддерживается и Си-подобная форма записи операции «равно» в виде двух знаков равенства `==`. Кроме того, доступны логические связки `&&` («и»), `||` («или») и `^^` («исключающее или»). Отметим, что все выражения, используемые в директиве `%if`, рассматриваются как *критические* (см. § 3.4.2).

Так же, как и для всех остальных `%if`-директив, для простого `%if` имеется форма сокращённой записи конструкции с `%else` — директива `%elif`.

Перечислим кратко остальные поддерживаемые NASM условные директивы. Директивы `%ifidn` и `%ifidni` принимают два аргумента, разделённые запятой, и сравнивают их как строки, предварительно произведя, если это необходимо, макроподстановки в тексте аргументов. Фрагмент кода, следующий за этими директивами, транслируется только в случае, если строки окажутся равными, причём `%ifidn` требует точного совпадения, тогда как `%ifidni` игнорирует регистр и считает, например, строки `foobar`, `FooBar` и `FOOBAR` одинаковыми. Для проверки противоположного условия можно использовать директивы `%ifnidn` и `%ifnidni`; все четыре директивы имеют `%elif`-формы, соответственно, `%elifidn`, `%elifidni`, `%elifnidn` и `%elifnidni`.

Директива `%ifmacro` проверяет существование многострочного макроса; поддерживаются директивы `%ifnmacro`, `%elifmacro` и `%elifnmacro`.

Директивы `%ifid`, `%ifstr` и `%ifnum` проверяют, является ли их аргумент, соответственно, идентификатором, строкой или числовой константой. Как обычно, NASM поддерживает все дополнительные формы вида `%ifnXXX`, `%elifXXX` и `%elifnXXX` для всех трёх директив.

Кроме перечисленных, NASM поддерживает директиву `%ifctx` и соответствующие формы, но объяснение её работы достаточно сложно и обсуждать эту директиву мы не будем.

## § 3.5.5. Макроповторения

При необходимости препроцессор NASM можно заставить многократно (циклически) обрабатывать один и тот же фрагмент кода. Это достигается директивами `%rep` (от слова *repetition*) и `%endrep`. Директива `%rep` принимает один обязательный параметр, означающий количество повторений. Фрагмент кода, заключённый между директивами `%rep` и `%endrep`, будет обработан препроцессором (и ассемблером) столько раз, сколько указано в параметре директивы `%rep`. Кроме того, между директивами `%rep` и `%endrep` может встретиться директива `%exitrep`, которая досрочно прекращает выполнение макроповторения.

Рассмотрим простой пример. Пусть нам необходимо описать область памяти, состоящую из 100 последовательных байтов, причём в первом из них должно содержаться число `50`, во втором — число `51` и т. д., в последнем, соответственно, число `149`. Конечно, можно просто написать сто строк кода:

```
db 50
db 51
db 52
;....
db 148
db 149
```

но это, во-первых, утомительно, а во-вторых, занимает слишком много места в тексте программы. Гораздо правильнее будет поручить генерацию этого кода макропроцессору, воспользовавшись макроповторением и макропеременной:

```
%assign n 50
%rep 100
        db n
%assign n n+1
%endrep
```

Встретив такой фрагмент, макропроцессор сначала свяжет с макропеременной `n` значение `50`, затем сто раз рассмотрит две строчки, заключённые между `%rep` и `%endrep`, причём каждое рассмотрение этих строк

приведёт к генерации очередной подлежащей ассемблированию строки `db 50`, `db 51`, `db 52` и т. д.; изменение числа происходит благодаря тому, что значение макропеременной `n` изменяется (увеличивается на единицу) на каждом проходе макроповторения. Иначе говоря, в результате обработки макропроцессором этого фрагмента как раз и получатся точно такие сто строк кода, как показано выше, и именно они и будут ассемблироваться. Макропроцессор, таким образом, избавляет нас от необходимости писать эти сто строк вручную.

Рассмотрим более сложный пример. Пусть имеется необходимость задать область памяти, содержащую последовательно в виде четырёхбайтных целых все числа Фибоначчи[7], не превосходящие 100000. Сгенерировать соответствующую последовательность директив `dd` можно с помощью такого фрагмента кода:

```
fibonacci
%assign i 1
%assign j 1
%rep 100000
  %if j > 100000
    %exitrep
  %endif

        dd j

  %assign k j+i
  %assign i j
  %assign j k
%endrep

fib_count equ ($-fibonacci)/4
```

причём метка `fibonacci` будет связана с адресом начала сгенерированной области памяти, а метка `fib_count` — с общим количеством чисел, размещённых в этой области памяти (с этим приёмом мы уже сталкивались на стр. 3.2).

Использовать макроповторения можно не только для генерации областей памяти, заполненных числами, но и для других целей. Пусть, например, у нас имеется массив из 128 двухбайтовых целых чисел:

[7] Напомним, что числа Фибоначчи — это последовательность чисел, начинающаяся с двух единиц, каждое следующее число которой получается сложением двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 41, 34 и т. д.

```
array   resw 128
```

и мы хотим написать последовательность из 128 команд `inc`, увеличивающих на единицу каждый из элементов этого массива. Можно сделать это так:

```
%assign a array
%rep 128
        inc word [a]
%assign a a+2
%endrep
```

Читатель мог бы отметить, что использование в такой ситуации 128 команд нерационально и правильнее было бы воспользоваться циклом во время исполнения, например, так:

```
        mov ecx, 128
lp:     inc word [array + ecx*2 - 2]
        loop lp
```

В большинстве случаев такой вариант действительно предпочтительнее, поскольку такие три команды, естественно, будут занимать в несколько десятков раз меньше памяти, чем последовательность из 128 команд `inc`, но следует иметь в виду, что работать такой код будет примерно в полтора раза медленнее, так что в некоторых случаях применение макроцикла для генерации последовательности одинаковых команд (вместо цикла времени исполнения) может оказаться осмысленным.

## § 3.5.6. Многострочные макросы и локальные метки

Вернёмся теперь к многострочным макросам; такие макросы генерируют не фрагмент строки, а фрагмент текста, состоящий из нескольких строк. Описание многострочного макроса также состоит из нескольких строк, заключённых между директивами `%macro` и `%endmacro`. В § 3.5.2 мы уже рассматривали простейшие примеры многострочных макросов, однако в мало-мальски сложном случае рассмотренных средств нам не хватит. Пусть, например, мы хотим описать макрос `zeromem`, принимающий на вход два параметра — адрес и длину области памяти — и раскрывающийся в код, заполняющий эту память нулями. Не особенно задумываясь над происходящим, мы могли бы написать, например, следующий (**неправильный!**) код:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
```

```
        push esi
        mov ecx, %2
        mov esi, %1
lp:     mov byte [esi], 0
        inc esi
        loop lp
        pop esi
        pop ecx
%endmacro
```

NASM примет такое описание и даже позволит произвести один макровызов. Если же в нашей программе встретятся хотя бы два вызова макроса `zeromem`, то при попытке оттранслировать программу мы получим сообщение об ошибке — NASM пожалуется на то, что мы используем одну и ту же метку (`lp:`) дважды. Действительно, при каждом макровызове макропроцессор вставит вместо вызова всё тело нашего макроопределения, только заменив `%1` и `%2` на соответствующие параметры, а всё остальное сохранив без изменения. Значит, строка

```
lp:     mov byte [esi], 0
```

содержащая метку `lp`, встретится ассемблеру (уже после макропроцессирования) дважды — или, точнее, ровно столько раз, сколько раз будет вызван макрос `zeromem`.

Ясно, что необходим некий механизм, позволяющий локализовать метку, используемую внутри многострочного макроса, с тем, чтобы такие метки, полученные вызовом одного и того же макроса в разных местах программы, не конфликтовали друг с другом. В NASM такой механизм называется «локальные метки в макросах». Чтобы задействовать этот механизм, необходимо начать имя метки с двух символов `%` — так, в приведённом выше примере оба вхождения метки `lp` нужно заменить на `%%lp`. Такая метка будет в каждом следующем макровызове заменяться некоторым новым (не повторяющимся) идентификатором. Отметим для наглядности, что при первом вызове макроса `zeromem` NASM заменит `%%lp` на `..@1.lp`, при втором — на `..@2.lp` и т. д.

Отметим ещё один недостаток вышеприведённого определения `zeromem`. Если при вызове этого макроса пользователь (программист, пользующийся нашим макросом, или, возможно, мы сами) использует в качестве первого параметра (адреса начала области памяти) регистр `ECX` или в качестве второго (длины области памяти) — регистр `ESI`, макровызов будет успешно оттранслирован, но работать программа будет совсем не так, как от неё ожидается. Действительно, если написать что-то вроде

```
section .bss
array   resb 256
arr_len equ $-array

section .text
; ...
        mov ecx, array
        mov esi, arr_len
        zeromem ecx, esi
; ...
```

то начало макроса zeromem развернётся в следующий код:

```
        push ecx
        push esi
        mov ecx, esi
        mov esi, ecx
        ; ...
```

в результате чего, очевидно, в **обоих** регистрах ECX и ESI окажется длина массива, а адрес его начала будет потерян. Скорее всего, программа в таком виде аварийно завершится, дойдя до этого фрагмента кода.

Чтобы избежать подобных проблем, можно воспользоваться директивами условной компиляции, проверяя, не является ли первый параметр регистром ECX и не является ли второй параметр регистром ESI, но можно поступить и проще — загрузить значения параметров в регистры через временную запись их в стек, то есть вместо

```
        mov ecx, %2
        mov esi, %1
```

написать

```
        push dword %2
        push dword %1
        pop esi
        pop ecx
```

Окончательно наше макроопределение примет следующий вид:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
        push esi
        push dword %2
        push dword %1
        pop esi
```

```
        pop ecx
%%lp:   mov byte [esi], 0
        inc esi
        loop %%lp
        pop esi
        pop ecx
%endmacro
```

## § 3.5.7. Макросы с переменным числом параметров

При описании многострочных макросов с помощью директивы `%macro` ассемблер NASM позволяет задать переменное число параметров. Это делается с помощью символа `-`, который в данном случае символизирует тире. Например, директива

```
%macro mymacro 1-3
```

задаёт макрос, принимающий от одного до трёх параметров, а директива

```
%macro mysecondmacro 2-*
```

задаёт макрос, допускающий произвольное количество параметров, не меньшее двух. При работе с такими макросами может оказаться полезным обозначение `%0`, вместо которого макропроцессор во время макроподстановки подставляет число, равное фактическому количеству параметров.

Напомним, что сами аргументы многострочного макроса в его теле обозначаются как `%1`, `%2` и т. д., но средств индексирования (то есть способа извлечь $n$-ый параметр, где $n$ вычисляется уже во время макроподстановки) NASM не предусматривает. Как же в таком случае использовать параметры, если даже их количество заранее не известно?

Проблему решает директива `%rotate`, позволяющая переобозначить параметры. Рассмотрим самый простой вариант директивы:

```
%rotate 1
```

Числовой параметр обозначает, на сколько позиций следует сдвинуть номера параметров. В данном случае это число 1, так что параметр, ранее обозначавшийся `%2`, после этой директивы будет иметь обозначение `%1`, в свою очередь бывший `%3` превратится в `%2` и т. д., ну а параметр, стоявший самым первым и имевший обозначение `%1`, в силу

«цикличности» нашего сдвига получит номер, равный общему количеству параметров. Обозначение %0 во всей этой процедуре не участвует и никак не изменяется.

Директива позволяет производить циклический сдвиг и в обратном направлении (влево), для этого следует задать её параметр отрицательным. Так, после отработки директивы

```
%rotate -1
```

%1 будет обозначать параметр, ранее стоявший самым последним, %2 станет обозначать параметр, ранее бывший первым (то есть имевший обозначение %1) и т. д.

Вспомним, что ранее (см. стр. 15) мы обещали написать макрос pcall, позволяющий в одну строчку сформировать вызов подпрограммы с любым количеством аргументов. Сейчас, имея в своём распоряжении макросы с переменным числом аргументов и директиву %rotate, мы готовы это сделать.

Наш макрос, который мы назовём просто pcall, будет принимать на вход адрес процедуры (аргумент для команды call) и произвольное количество параметров, предназначенное для размещения в стеке. Мы будем, как и раньше, предполагать для простоты, что каждый параметр занимает ровно 4 байта. Напомним, что параметры должны быть помещены в стек в обратном порядке, начиная с последнего. Мы добьёмся этого с помощью макроцикла %rep и директивы %rotate -1, которая на каждом шаге будет делать последний (на текущий момент) параметр параметром номер 1. Количество итераций цикла на единицу меньше, чем количество параметров, переданных в макрос, потому что первый из параметров является именем процедуры и его в стек заносить не надо. После этого цикла нам останется снова превратить последний параметр в первый (на этот раз это как раз и окажется самый первый из всех параметров, то есть адрес процедуры) и сделать call, а затем вставить команду add для очистки стека от параметров. Итак, пишем:

```
%macro pcall 1-*  ; от одного до скольки угодно
  %rep %0 - 1     ; цикл по всем параметрам кроме первого
    %rotate -1    ; последний параметр становится %1
        push dword %1
  %endrep
  %rotate -1      ; адрес процедуры становится %1
        call %1
        add esp, (%0 - 1) * 4
%endmacro
```

Если теперь вызвать этот макрос, например, вот так:

```
pcall myproc, eax, myvar, 27
```

то результатом подстановки станет следующий фрагмент:

```
push dword 27
push dword myvar
push dword eax
call myproc
add esp, 12
```

что, собственно, нам и требовалось.

## § 3.5.8. Макродирективы для работы со строками

Ассемблер NASM поддерживает две директивы, предназначенные для преобразования строк (строковых констант) во время макропроцессирования. Они могут оказаться полезными, например, внутри многострочного макроса, одним из параметров которого является (должна быть) строка и с этой строкой необходимо предварительно выполнить те или иные преобразования.

Первая из директив, `%strlen`, позволяет определить длину строки. Директива имеет два параметра. Первый из них — имя макропеременной, которой следует присвоить число, соответствующее длине строки, ну а второй — собственно строка. Так, в результате выполнения

```
%strlen sl 'my string'
```

макропеременная `sl` получит значение 9.

Вторая директива, `%substr`, позволяет выделить из строки символ с заданным номером. Например, после выполнения

```
%substr var1 'abcd' 1
%substr var2 'abcd' 2
%substr var3 'abcd' 3
```

макропеременные `var1`, `var2` и `var3` получат значения `'a'`, `'b'` и `'c'` соответственно, то есть произойдёт то же самое, как если бы мы написали

```
%define var1 'a'
%define var2 'b'
%define var3 'c'
```

Всё это имеет смысл, как правило, только в случае, если в качестве аргумента директивы получают либо имя макропеременной, либо обозначение позиционного параметра в многострочном макросе.

Напомним, что все макродирективы отрабатывают во время макропроцессирования (**перед** компиляцией, то есть задолго до выполнения нашей программы), так что, разумеется, на момент соответствующих макроподстановок все используемые строки должны быть уже известны.

## § 3.6. Командная строка NASM

Рассказ об ассемблере NASM мы завершаем кратким обзором аргументов его командной строки. Как уже говорилось, при вызове программы `nasm` необходимо указать имя файла, содержащего исходный текст на языке ассемблера, а кроме этого, обычно требуется указать *ключи*, задающие режим работы. С некоторыми из них мы уже знакомы: это флаги `-f`, `-o` и `-d`.

Напомним, что ключ `-f` позволяет указать *формат* получаемого кода. В нашем случае всегда используется формат `elf`. Интересно, что, если не указать этот флаг, ассемблер создаст выходной файл в «сыром» формате, то есть, попросту говоря, переведёт наши команды в двоичное представление и в таком виде запишет в файл. Работая под управлением операционных систем, мы такой файл не запустить на выполнение не сможем, однако если бы мы, к примеру, хотели написать программу для размещения в загрузочном секторе диска, то «сырой» формат оказался бы как раз тем, что нам нужно.

Ключ `-o` задаёт имя файла, в который следует записать результат трансляции. Если мы используем формат `elf`, то вполне можем доверить выбор имени файла самому NASM'у: он отбросит от имени исходного файла суффикс `.asm` и заменит его на `.o`, что нам в большинстве случаев и требуется. Если же по каким-то причинам нам удобнее другое имя, мы можем указать его явно с помощью `-o`.

Ключ `-d`, как мы уже знаем (см. стр. 22), используется для определения макросимвола в случае, если мы не хотим делать этого путём редактирования исходного текста. Мы использовали его в форме `-dSYMBOL`, что даёт тот же эффект, как если в начало программы вставить строку `%define SYMBOL`. Но можно использовать его и для задания *значения* макросимвола: например, `-dSIZE=1024` не только определит символ

SIZE, но и припишет ему значение 1024, как это сделала бы директива %define SIZE 1024.

Очень интересные в познавательном плане возможности даёт ключ -l, после которого требуется указать имя файла. Этот ключ включает генерацию так называемого ***листинга*** — подробного отчёта ассемблера о проделанной работе. Листинг включает в себя строки исходного кода, снабжённые информацией об используемых адресах и о том, какой итоговый код сгенерирован в результате обработки каждой исходной строки. Для примера возьмите любую программу на языке ассемблера и оттранслируйте её с флагом -l; так, если ваша программа называется prog.asm, попробуйте применить команду

```
nasm -f elf -l prog.lst prog.asm
```

в результате которой текст листинга будет помещён в файл prog.lst; обязательно просмотрите получившийся файл и если в его содержании что-то осталось непонятным, задайте вопросы вашему преподавателю.

Весьма полезным может оказаться ключ -g, указывающий NASM'у на необходимость включения в результаты трансляции так называемой *отладочной информации*. При указании этого ключа NASM вставляет в объектный файл помимо объектного кода ещё и сведения об имени исходного файла, номерах строк в нём и т. п. Для работы программы вся эта информация совершенно бесполезна, тем более что по объёму она может в несколько раз превышать «полезный» объектный код. Однако в случае, если ваша программа работает не так, как вы от неё ожидаете, компиляция с флажком -g позволит вам воспользоваться отладчиком (например, gdb) для пошагового выполнения программы, что, в свою очередь, даст возможность разобраться в происходящем.

Ещё один полезный ключ — -e; он предписывает NASM'у прогнать наш исходный код через макропроцессор, выдать результат в поток стандартного вывода (попросту говоря, на экран) и на этом успокоиться. Такой режим работы может оказаться полезен, если мы ошиблись при написании макроса и никак не можем понять, в чём наша ошибка заключается; увидев результат макропроцессирования нашей программы, мы, скорее всего, сможем понять, что и почему пошло не так.

NASM поддерживает и другие ключи командной строки; желающие могут изучить их самостоятельно, обратившись к документации.

# Глава 4. Взаимодействие с операционной системой

В этой главе мы рассмотрим средства взаимодействия пользовательской программы с операционной системой, что позволит в дальнейшем отказаться от использования макросов из файла `stud_io.inc`, а при желании и самостоятельно создавать их аналоги.

Пользовательские задачи обращаются к ядру операционной системы, используя так называемые ***системные вызовы***, которые, в свою очередь, реализованы через механизм ***программных прерываний***. Чтобы понять, что это такое, нам придётся подробно обсудить, что такое прерывания, какие они бывают и для чего служат, поэтому первые два параграфа этой главы мы посвятим изложению необходимых теоретических сведений, и лишь затем, имея готовую базу, рассмотрим механизм системных вызовов операционных систем Linux и FreeBSD на уровне машинных команд.

## § 4.1. Мультизадачность и её основные виды

### § 4.1.1. Понятие одновременности выполнения

Как уже говорилось во введении, ***мультизадачность*** или режим мультипрограммирования — это такой режим работы вычислительной системы, при котором несколько программ могут выполняться в системе одновременно. Следует отметить, что для этого, вообще говоря, не нужны несколько физических *процессоров*. Вычислительная система может иметь всего один процессор, что не мешает само по себе реализации режима мультипрограммирования. Так или иначе, количество про-

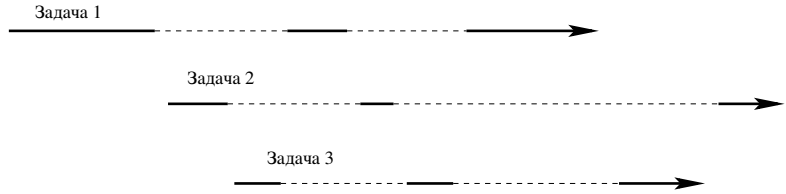


Рис. 4.1. Одновременное выполнение задач на одном процессоре

цессоров в системе в общем случае меньше количества одновременно выполняемых программ.

Ясно, что процессор в каждый момент времени может выполнять только одну программу. Что же, в таком случае, понимается под мультипрограммированием? Кажущийся парадокс разрешается введением следующего определения ***одновременности*** для случая выполняющихся программ (***процессов***, или ***задач***):

**Две задачи, запущенные на одной вычислительной системе, называются выполняемыми *одновременно*, если периоды их выполнения (временно́й отрезок с момента запуска до момента завершения каждой из задач) полностью или частично перекрываются.**

Итак, если процессор, работая в каждый момент времени с одной задачей, при этом переключается между несколькими задачами, уделяя внимание то одной из них, то другой, эти задачи в соответствии с нашим определением будут считаться выполняемыми одновременно (см. рис. 4.1).

## § 4.1.2. Пакетный режим

В простейшем случае мультизадачность позволяет решить проблему простоя центрального процессора во время операций ввода-вывода. Представим себе вычислительную систему, в которой выполняется одна задача (например, обсчет сложной математической модели). В некоторый момент времени задаче может потребоваться операция обмена данными с каким-либо внешним устройством (например, чтение очередного блока входных данных либо, наоборот, запись конечных или промежуточных результатов).

Скорость работы внешних устройств (лент, магнитных барабанов, дисков и т. п.) обычно на порядки ниже, чем скорость работы централь-

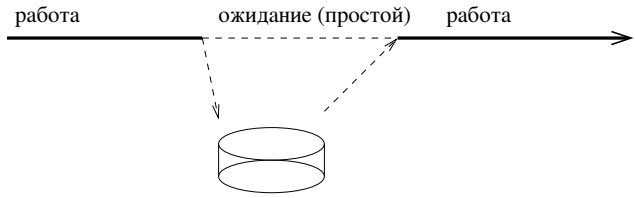


Рис. 4.2. Простой процессора в однозадачной системе

ного процессора, и в любом случае никоим образом не бесконечна. Так, для чтения заданного блока данных с диска необходимо включить привод головки, чтобы переместить её в нужное положение (на нужную дорожку) и дождаться, пока сам диск повернётся на нужный угол (для работы с заданным сектором); затем, пока сектор проходит под головкой, прочитать записанные в этом секторе данные во внутренний буфер контроллера диска[1]; наконец, следует разместить прочитанные данные в той области памяти, где их появления ожидает пользовательская программа, и лишь после этого вернуть ей управление.

Всё это время (как минимум, время, затрачиваемое на перемещение головки и ожидание нужной фазы поворота диска) центральный процессор будет простаивать (рис. 4.2). Если задача у нас всего одна и больше делать нечего, такой простой не создаёт проблем, но если кроме той задачи, которая уже работает, у нас есть и другие задачи, дожидающиеся своего часа, то лучше бы было употребить время центрального процессора, впустую пропадающее в ожидании окончания операций ввода-вывода, на решение других задач.

Именно так поступают мультизадачные операционные системы. В такой системе из задач, которые нужно решать, формируется *очередь заданий*. Как только активная задача затребует проведение операции ввода-вывода, операционная система выполняет необходимые действия по запуску контроллеров устройств на исполнение запрошенной операции либо ставит запрошенную операцию в очередь, если начать её немедленно по каким-то причинам нельзя, после чего активная задача заменяется на другую — новую (взятую из очереди) или уже выполнявшуюся раньше, но не успевшую завершиться. Замененная задача в этом случае считается перешедшей в состояние ожидания результата ввода-вывода, или ***состояние блокировки***.

[1] Чтение непосредственно в оперативную память теоретически возможно, но технически сопряжено с определенными трудностями и применяется редко

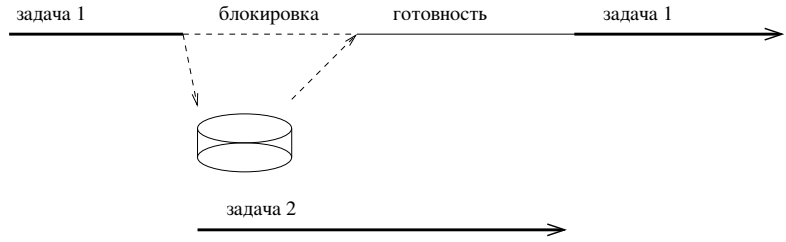


Рис. 4.3. Пакетная ОС

В простейшем случае новая активная задача остается в режиме выполнения до тех пор, пока она не завершится либо не затребует, в свою очередь, проведение операции ввода-вывода. При этом блокированная задача по окончании операции ввода-вывода переходит из состояния блокировки в ***состояние готовности к выполнению***, но переключения на нее не происходит (см. рис. 4.3); это обусловлено тем, что операция смены активной задачи, вообще говоря, отнимает много процессорного времени.

Такой способ построения мультизадачности, при котором смена активной задачи происходит только в случае ее окончания или запроса на операцию ввода-вывода, называется ***пакетным режимом***[2], а операционные системы, реализующие этот режим,— ***пакетными операционными системами***.

Режим пакетной мультизадачности является самым эффективным с точки зрения использования вычислительной мощности центрального процессора, поэтому именно пакетный режим используется для управления суперкомпьютерами и другими машинами, основное назначение которых — большие объемы численных расчетов.

## § 4.1.3. Режим разделения времени

С появлением первых терминалов и диалогового (иначе говоря, интерактивного) режима работы с компьютерами возникла потребность в

[2]Русскоязычный термин ≪пакетный режим≫ является устоявшимся, хотя и не слишком удачным переводом английского термина ≪batch mode≫; слово *batch* можно также перевести как ≪колода≫ (собственно, изначально имелись в виду колоды перфокарт, олицетворявшие задания). Не следует путать этот термин со словами, происходящими от английского слова packet, которое тоже обычно переводится на русский как ≪пакет≫.

других стратегиях смены активных задач, или, как принято говорить, ***планирования времени центрального процессора***.

Действительно, пользователю, ведущему диалог с той или иной программой, вряд ли захочется ждать, пока некая активная задача, вычисляющая, скажем, обратную матрицу порядка 1000x1000, завершит свою работу. При этом много процессорного времени на обслуживание диалога с пользователем не требуется: в ответ на каждое действие пользователя (например, нажатие на клавишу) обычно необходимо выполнить набор действий, укладывающийся в несколько миллисекунд, тогда как самих таких событий пользователь даже в режиме активного набора текста может создать никак не больше трех-четырех в секунду (скорость компьютерного набора 200 символов в минуту считается очень высокой). Соответственно, было бы нелогично ждать, пока пользователь полностью завершит свой диалоговый сеанс: бóльшую часть времени процессор мог бы производить арифметические действия, необходимые для задачи, вычисляющей матрицу.

Решить проблему позволяет ***режим разделения времени***. В этом режиме каждой задаче отводится определенное время работы, называемое ***квантом времени***. По окончании этого кванта, если в системе имеются другие готовые к исполнению задачи, активная задача принудительно приостанавливается и заменяется другой задачей. Приостановленная задача помещается в ***очередь задач, готовых к выполнению*** и находится там, пока остальные задачи отработают свои кванты; затем она снова получает очередной квант времени для работы, и т. д.

Естественно, если активная задача затребовала операцию ввода-вывода, она переводится в состояние блокировки (точно так же, как и в пакетном режиме). Задачи, находящиеся в состоянии блокировки, не ставятся в очередь на выполнение и не получают квантов времени до тех пор, пока операция ввода-вывода не будет завершена (либо не исчезнет другая причина блокировки), и задача не перейдет в состояние готовности к выполнению.

Отметим, что существуют различные алгоритмы поддержки очереди на выполнение, в том числе и такие, в которых задачам приписывается некоторый приоритет, выраженный числом. Например, в ОС Unix обычно задача имеет две составляющие приоритета — статическую и динамическую; статическая составляющая представляет собой заданный администратором уровень «важности» выполнения данной конкретной задачи, динамическая же изменяется планировщиком: пока задача находится в стадии выполнения, её динамический приоритет падает, когда же задача находится в очереди на исполнение, динамическая составля-

ющая приоритета, напротив, растёт. Из нескольких готовых к исполнению задач выбирается имеющая наибольшую сумму приоритетов, так что рано или поздно задача даже с самым низким статическим приоритетом получит управление за счет возросшего динамического приоритета.

Некоторые операционные системы, включая ранние версии Windows, применяли стратегию, занимающую промежуточное положение между пакетным режимом и режимом разделения времени. В этих системах задачам выделялся квант времени, как и в системах разделения времени, но принудительной смены текущей задачи по истечении кванта времени не производилось; система проверяла, не истек ли квант времени у текущей задачи, только когда задача обращалась к операционной системе за какими-либо услугами (не обязательно за вводом-выводом). Таким образом, задача, не нуждающаяся в услугах операционной системы, могла оставаться на процессоре сколь угодно долго, как и в пакетных операционных системах. Такой режим работы называется ***невытесняющим***. В современных системах он не применяется, поскольку налагает слишком жесткие требования на исполняемые в системе программы; так, в ранних версиях Windows любая программа, занятая длительными вычислениями, блокировала работу всей системы.

## § 4.1.4. Режим реального времени

Иногда режим разделения времени также оказывается непригоден. В некоторых ситуациях, таких как управление полетом самолета, ядерным реактором, автоматической линией производства и т. п., некоторые задачи должны быть завершены строго до определенного момента времени; так, если автопилот самолета, получив сигнал от датчиков тангажа и крена, потратит на вычисление необходимого корректирующего воздействия больше времени, чем допустимо, самолет может вовсе потерять управление.

В случае, когда выполняемые задачи (как минимум некоторые из них) имеют жесткие рамки по необходимому времени завершения, применяются ***операционные системы реального времени***. В отличие от систем разделения времени, задача планировщика реального времени не в том, чтобы дать всем программам отработать некоторое время, а в том, чтобы *обеспечить завершение каждой задачи за отведённое ей время*, если же это невозможно — снять задачу, освободив процессор для тех задач, которые ещё можно успеть завершить к сроку.

## § 4.1.5. Аппаратная поддержка мультизадачности

Ясно, что для построения мультизадачного режима работы вычислительной системы аппаратура (прежде всего сам центральный процессор) должна обладать определенными свойствами. О некоторых из

них мы уже говорили в § 1.2 — это, во-первых, защита памяти, а во-вторых, разделение машинных команд на обычные и привилегированные, с отключением возможности выполнения привилегированных команд в ограниченном режиме работы центрального процессора.

Действительно, при одновременном нахождении в памяти машины нескольких программ, если не предпринять специальных мер, одна из программ может модифицировать данные или код других программ или самой операционной системы. Даже если допустить отсутствие злого умысла у разработчиков всех запускаемых программ, от случайных ошибок в программах нас это допущение не спасет, причём такая ошибка может, с одной стороны, привести к тяжелым авариям всей системы, а с другой стороны — оказаться совершенно неуловимой, вплоть до абсолютной невозможности установить, какая из задач «виновата» в происходящем. Дело в том, что для обнаружения и устранения ошибки необходима возможность более-менее надёжно воссоздавать обстоятельства, при которых эта ошибка проявляется, а точно воссоздать состояние всей системы со всеми запущенными в ней задачами практически невозможно.

Итак, необходимы средства ограничения возможностей работающей программы по доступу к областям памяти, занятым другими программами. Программно такую защиту можно реализовать разве что путем интерпретации всего машинного кода исполняющейся программы, что, как правило, недопустимо из соображений эффективности. Таким образом, необходима *аппаратная* поддержка защиты памяти, позволяющая ограничить возможности текущей задачи по доступу к оперативной памяти.

Коль скоро существует защита памяти, процессор должен иметь набор команд для управления этой защитой. Если, опять таки, не предпринять специальных мер, то такие команды сможет исполнить любая из выполняющихся программ, сняв защиту памяти или модифицировав ее конфигурацию, что сделало бы саму защиту памяти практически бессмысленной.

Рассматриваемая проблема касается не только защиты памяти, но и работы с внешними устройствами. Как уже говорилось, чтобы обеспечить нормальное взаимодействие всех программ с устройствами ввода-вывода, операционная система должна взять непосредственную работу с устройствами на себя, а пользовательским программам предоставлять интерфейс для обращения к операционной системе за услугами по работе с устройствами. Иначе говоря, пользовательские программы должны иметь возможность работы с внешними устройствами только через

операционную систему. Соответственно, необходимо запретить пользовательским программам выполнение команд процессора, осуществляющих чтение/запись портов ввода-вывода.

Вообще, передавая управление пользовательской программе, операционная система должна быть уверена, что задача не сможет (иначе как путем обращения к самой операционной системе) выполнить никакие действия, влияющие на систему в целом.

Проблема решается введением двух режимов работы центрального процессора: ***привилегированного*** и ***ограниченного***. В литературе привилегированный режим часто называют «режимом ядра» или «режимом супервизора» (англ. ***kernel mode***, ***supervisor mode***). Ограниченный режим называют также «пользовательским режимом» (англ. ***user mode***) или просто ***непривилегированным*** (англ. ***nonprivileged***). Термин ***ограниченный режим*** избран в нами как наиболее точно описывающий сущность этого режима работы центрального процессора без привязки к его использованию операционными системами. В привилегированном режиме процессор может выполнять любые существующие команды. В ограниченном режиме выполнение команд, влияющих на систему в целом, запрещено; разрешаются только команды, эффект которых ограничен модификацией данных в областях памяти, не закрытых защитой памяти. Сама операционная система выполняется в привилегированном режиме, пользовательские программы — в ограниченном.

Как мы уже отмечали в § 1.2, пользовательская программа может только модифицировать данные в отведённой ей памяти; любые другие действия требуют обращения к операционной системе. Это обеспечивается поддержкой в центральном процессоре механизма защиты памяти и наличием ограниченного режима работы. Соблюдения этих двух аппаратных требований, однако, ещё не достаточно для реализации мультизадачного режима работы системы.

Вернемся к ситуации с операцией ввода-вывода. В однозадачной системе (рис.4.2 на стр. 36) во время исполнения операции ввода-вывода центральный процессор мог непрерывно опрашивать контроллер устройства на предмет его готовности (завершена ли требуемая операция), после чего произвести необходимые действия по подготовке к возобновлению работы активной задачи — в частности, скопировать прочитанные данные из буфера контроллера в область памяти, в которой задача ожидает появления данных. Следует отметить, что в этом случае процессор был бы непрерывно занят во время операции ввода-вывода, несмотря на то, что никаких полезных вычислений он

при этом не производил. Такой способ взаимодействия называется ***активным ожиданием***. Ясно, что активное ожидание неэффективно, так как процессорное время можно было бы использовать с бóльшей пользой.

При переходе к мультизадачной обработке, показанной на рис. 4.3 на стр. 37, возникает определенная проблема. В момент завершения операции ввода-вывода процессор занят исполнением второй задачи. Между тем, в момент завершения операции требуется как минимум перевести первую задачу из состояния блокировки в состояние готовности; более того, могут потребоваться и другие действия, такие как копирование данных из буфера контроллера, сброс контроллера (например, выключение мотора диска), а в более сложных ситуациях — инициирование другой операции ввода-вывода, ранее отложенной (это может быть операция чтения с того же диска, которую затребовала другая задача в то время, как первая операция еще выполнялась). Проблема состоит в том, каким образом операционная система узнает о завершении операции ввода-вывода, если процессор при этом занят выполнением другой задачи и непрерывного опроса контроллера не производит.

Решить проблему позволяет аппарат ***прерываний***. В данном конкретном случае в момент завершения операции контроллер подает центральному процессору определенный сигнал (электрический импульс), называемый ***запросом прерывания***. Центральный процессор, получив этот сигнал, прерывает выполнение активной задачи и передает управление процедуре операционной системы, которая выполняет все действия, необходимые по окончании операции ввода-вывода. Такая процедура называется ***обработчиком прерывания***. После завершения процедуры-обработчика управление возвращается активной задаче.

Для реализации пакетного мультизадачного режима достаточно, чтобы на уровне аппаратуры были реализованы прерывания, защита памяти и два режима работы. Если же необходимо реализовать систему разделения времени или реального времени, требуется наличие в аппаратуре еще одного компонента — ***таймера***.

Действительно, планировщику операционной системы разделения времени нужна возможность отслеживания истечения квантов времени, выделенных пользовательским программам; в системе реального времени такая возможность также необходима, причем требования к ней даже более жёсткие: не сняв вовремя с процессора активное на тот момент приложение, планировщик рискует не успеть выделить более важным программам необходимое им процессорное время, в результа-

те чего могут наступить неприятные последствия (вспомните пример с автопилотом самолёта).

Таймер представляет собой сравнительно простое устройство, вся функциональность которого сводится к генерации прерываний через равные промежутки времени. Эти прерывания дают возможность операционной системе получить управление, проанализировать текущее состояние имеющихся задач и при необходимости сменить активную задачу.

Итак, для реализации мультизадачной операционной системы аппаратное обеспечение компьютера обязано поддерживать:

- аппарат прерываний;
- защиту памяти;
- привилегированный и ограниченный режимы работы центрального процессора;
- таймер.

Первые три свойства необходимы в любой мультизадачной системе, последнее может отсутствовать в случае пакетной планировки (хотя в реально существующих системах таймер присутствует всегда). Следует обратить внимание, что из перечисленных свойств только таймер является отдельным устройством, остальные три представляют собой особенности центрального процессора.

## § 4.2. Виды прерываний

Современный термин «***прерывание***» довольно далеко ушел в своем развитии от изначального значения; начинающие программисты часто с удивлением обнаруживают, что некоторые прерывания вовсе ничего не прерывают.

Дать строгое определение прерывания было бы несколько затруднительно. Вместо этого попытаемся объяснить сущность различных видов прерываний и найти между ними то общее, что и оправдывает существование самого термина.

### § 4.2.1. Внешние (аппаратные) прерывания

Прерывания в изначальном смысле уже знакомы нам из предыдущего параграфа. Те или иные устройства вычислительной системы могут

осуществлять свои функции независимо от центрального процессора; в этом случае им может время от времени требоваться внимание операционной системы, но единственный центральный процессор (или, что ничуть не лучше, все имеющиеся в системе центральные процессоры) может быть именно в такой момент занят обработкой пользовательской программы.

Аппаратные (или ***внешние***) прерывания были призваны решить эту проблему. Для поддержки аппаратных прерываний процессор имеет специально предназначенные для этого контакты; электрический импульс, поданный на такой контакт, воспринимается процессором как сигнал о том, что некоторому устройству требуется внимание операционной системы. В современных архитектурах, основанных на общей шине, для запроса прерывания используется одна из дорожек шины.

Последовательность событий при возникновении и обработке прерывания выглядит приблизительно следующим образом[3]:

1. Устройство, которому требуется внимание процессора, устанавливает на шине сигнал «запрос прерывания».

2. Процессор доводит выполнение текущей программы до такой точки, в которой выполнение можно прервать так, чтобы потом восстановить его с того же места; после этого процессор выставляет на шине сигнал «подтверждение прерывания». При этом другие прерывания блокируются.

3. Получив подтверждение прерывания, устройство передает по шине некоторое число, идентифицирующее данное устройство; это число называют ***номером прерывания***.

4. Процессор сохраняет где-то (обычно на стеке) текущие значения счетчика команд и регистра флагов; это называется ***малым упрятыванием***. Счетчик команд и регистр флагов должны быть сохранены по той причине, что выполнение первой же инструкции обработчика прерывания изменит (испортит) и то, и другое, сделав невозможным прозрачный (т. е. незаметный для пользовательской задачи) возврат из обработчика; остальные регистры обработчик прерывания может при необходимости сохранить самостоятельно.

5. Устанавливается привилегированный режим работы центрального процессора, после чего управление передается на точку входа

[3]Здесь приводится общая схема; в действительности все намного сложнее.

процедуры в операционной системе, называемой, как мы уже говорили, ***обработчиком прерывания***. Адрес обработчика может быть предварительно считан из специальных областей памяти, либо вычислен иным способом.

Напомним, что переключение из привилегированного режима работы центрального процессора в ограниченный можно осуществить простой командой, поскольку в привилегированном режиме доступны все возможности процессора; в то же время, переход из ограниченного (пользовательского) режима обратно в привилегированный произвести с помощью обычной команды нельзя, поскольку это лишило бы смысла само существование привилегированного и ограниченного режимов. В этом плане **прерывание интересно ещё и тем, что при его возникновении режим работы центрального процессора становится привилегированным**.

## § 4.2.2. Внутренние прерывания (ловушки)

Чтобы понять, о чем пойдет речь в этом параграфе, рассмотрим следующий вопрос: что следует делать центральному процессору, если активная задача выполнила целочисленное деление на ноль?

Ясно, что дальнейшее выполнение программы лишено смысла: результат деления на ноль невозможно представить каким-либо целым числом, так что в переменной, которая должна была содержать результат произведённого деления, в лучшем случае будет содержаться мусор; соответственно, и конечные результаты, скорее всего, окажутся иррелевантными.

Пытаться оповестить программу о происшедшем путем выставления какого-нибудь флага, очевидно, также бессмысленно. Если программист не произвел **перед** выполнением деления проверку делителя на равенство нулю, представляется и вовсе ничтожной вероятность того, что он станет проверять **после** деления значение какого-то флага.

Завершить текущую задачу процессор самостоятельно не может. Это слишком сложное действие, зависящее от реализации операционной системы. Остается только один вариант: передать управление операционной системе, известив её о происшедшем. Что делать с аварийной задачей, операционная система решит самостоятельно.

Отметим, что требуется, вообще говоря, переключиться в привилегированный режим и передать управление коду операционной системы; перед этим желательно сохранить регистры (хотя бы счетчик команд и

регистр флагов); даже если задача ни при каких условиях не будет продолжена с того же места (а предполагать это процессор, вообще говоря, не вправе), значения регистров в любом случае пригодятся операционной системе для анализа происшествия. Более того, каким-то образом следует сообщить операционной системе о причине того, что управление передано ей; кроме деления на ноль, такими причинами могут быть нарушение защиты памяти, попытка выполнить запрещенную или несуществующую инструкцию, попытка прочитать слово по нечетному адресу и т. п.

Легко заметить, что действия, которые должен выполнить процессор, оказываются очень похожи на рассмотренный ранее случай аппаратного прерывания. Основное отличие состоит в отсутствии обмена по шине (запроса и подтверждения прерывания): действительно, информация о перечисленных событиях возникает внутри процессора, а не вне его[4]. Остальные шаги по обработке деления на ноль и других подобных ситуаций повторяют шаги по обработке аппаратного прерывания практически дословно.

Поэтому обработку ситуаций, в которых дальнейшее выполнение активной задачи оказывается невозможной по причине выполненных ею некорректных действий, называют так же, как и действия по запросу внешних устройств — прерываниями. Чтобы не путать разные по своей природе прерывания, их делят на внешние (аппаратные) и внутренние; такая терминология оправдана тем, что причина внешнего прерывания находится вне центрального процессора, тогда как причина внутреннего — у ЦП внутри. Иногда внутренние прерывания называют иначе, например ***ловушками*** (traps) или как-то еще.

## § 4.2.3. Программные прерывания

Как уже говорилось, пользовательской задаче не позволяется делать ничего, кроме преобразования данных в отведенной ей памяти. Все действия, затрагивающие внешний по отношению к задаче мир, выполняются через операционную систему. Соответственно, необходим механизм, позволяющий пользовательской задаче обратиться к ядру операционной системы за теми или иными услугами. Напомним, что **обращение пользовательской задачи к ядру операционной системы за услугами называется *системным вызовом***.

[4] С точки зрения реализации внутренние прерывания могут оказаться многократно проще, чем аппаратные, за счет того, что они всегда происходят на определенной фазе выполнения инструкции; подробности читатель найдет в книге [3].

Ясно, что по своей сути системный вызов — это передача управления от пользовательской задачи ядру операционной системы. Однако здесь есть две проблемы. Во-первых, ядро работает в привилегированном режиме, а пользовательская задача — в ограниченном. Во-вторых, пространство адресов ядра для пользовательской задачи обычно недоступно (более того, в адресном пространстве задачи этих адресов может вообще не быть). Впрочем, даже если бы оно было доступно, позволить пользовательской задаче передавать управление в произвольную точку ядра было бы несколько странно.

Таким образом, для осуществления системного вызова необходимо сменить режим выполнения с пользовательского на привилегированный и передать управление в некоторую точку входа в операционной системе. Нам уже известны два случая, в которых происходит что-то подобное — это аппаратные и внутренние прерывания. Изобретать дополнительный механизм для системного вызова не обязательно: для его реализации можно использовать частный случай внутреннего прерывания, инициируемый специально предназначенной для этого машинной инструкцией. На разных архитектурах соответствующая инструкция может называется `trap` (ловушка), `svc` (supervisor call, то есть «обращение к супервизору» и т. д. Рассматриваемые нами процессоры семейства i386 используют команду `int` (от слова interrupt — прерывание). Такое прерывание называется ***программным прерыванием***. Отличие этого вида прерывания от остальных состоит в том, что оно происходит по инициативе пользовательской задачи, тогда как другие прерывания случаются без её ведома: внешние — по требованию внешних устройств, внутренние — в случае непредвиденных обстоятельств, которые вряд ли были выполняемой программой предусмотрены.

Некоторые авторы не делают различия между терминами «программное прерывание» и «системный вызов», называя системным вызовом как само обращение к ОС, так и программное прерывание, используемое для его осуществления.

Некоторые процессоры могут предусматривать и иные механизмы передачи управления операционной системе. Так, процессоры семейства i386 реализуют так называемые *шлюзы* (англ. gates) для передачи управления привилегированным программам с одновременным повышением уровня привилегированности режима работы процессора, а самих этих уровней, называемых *кольцами защиты*, процессоры семейства i386 поддерживают не два, а четыре; впрочем, операционные системы этим обычно не пользуются.

Так или иначе, повышение уровня привилегий (переход из ограниченного режима в привилегированный) возможно только при условии одновременной передачи управления на заранее заданную точку входа, причем адреса возможных точек входа могут настраиваться только в привилегированном режиме. Таким образом, операционная система имеет возможность гарантировать, что при смене режима работы на привилегированный управление получит только код самой операционной системы, причем только такой её код, который для этого специально предназначен. Иначе говоря, исполнение в привилегированном режиме какого бы то ни было пользовательского кода полностью исключается.

## § 4.3. Системные вызовы в ОС Unix

Перейдём теперь к освоению системных вызовов на практике. Следует отметить, что соглашения о том, как конкретно должен происходить системный вызов, как передать ему необходимые параметры, какое использовать прерывание, как получить результат выполнения и т. п., варьируются от системы к системе. Даже если речь идёт о двух представителях семейства Unix (ОС GNU/Linux и ОС FreeBSD), работающих на одной и той же аппаратной платформе i386, низкоуровневая реализация системных вызовов оказывается в них совершенно различна. Следующие два параграфа будут посвящены описанию соглашений об организации системных вызовов этих двух систем[5]; при желании вы можете прочитать только один из этих двух параграфов, относящийся к той системе, которую вы используете.

Следует иметь в виду, что системы семейства Unix расчитаны в основном на программирование на языке Си. Естественно, для этого языка вместе с системой поставляются библиотеки, облегчающие работу с системными вызовами — в частности, для каждого системного вызова предоставляется библиотечная функция, позволяющая обратиться к услугам ядра как к обычной подпрограмме. Системные вызовы в ОС Unix имеют названия, совпадающие с именами соответствующих функций-обёрток из библиотеки языка Си.

К сожалению, такая ориентированность на Си приводит к некоторым неудобствам при работе на уровне языка ассемблера. Так, некоторые системные вызовы при переходе от системы к системе могут менять свои номера (например, `getppid` в ОС GNU/Linux имеет номер 64, а в

[5]Естественно, ОС GNU/Linux рассматривается в варианте для i386; версии этой системы, предназначенные для других аппаратных архитектур, устроены иначе.

ОС FreeBSD — номер 39). Программисты, работающие на языке Си, об этом могут не задумываться, поскольку в любой системе семейства Unix им достаточно вызвать обычную функцию с именем `getppid`, а конкретное исполнение системного вызова возлагается на библиотеку, которая прилагается к системе, так что программа, написанная программистом на Си с использованием `getppid`, будет успешно компилироваться на любой системе и работать одинаково. Иное дело, если мы пишем на языке ассемблера. Никакой библиотеки системных вызовов у нас при этом нет, номер вызова мы должны указать в программе явно, так что в тексте, предназначенном для GNU/Linux, придётся использовать число `64`, тогда как для FreeBSD нужно будет число `39`. Получается, что написанный нами исходный текст будет пригоден для одной системы и ошибочен для другой. Аналогично обстоят дела и с некоторыми числовыми константами, которые вызовы получают на вход.

Частично нас может выручить макропроцессор с его директивами условной компиляции, либо мы можем ограничиться только одной системой (что, на самом деле, не совсем правильно). К счастью, системы FreeBSD и GNU/Linux всё же во многом похожи друг на друга и числовые значения, связанные с системными вызовами, много где попросту совпадают (с другими системами семейства Unix было бы хуже). Так или иначе, кто предупреждён, тот вооружён.

### § 4.3.1. Конвенция ОС GNU/Linux

Ядро Linux на платформе i386 использует для осуществления системного вызова прерывание с номером `80h`. Номер системного вызова передаётся ядру через регистр `EAX`; если системный вызов принимает параметры, то они располагаются, соответственно, в регистрах `EBX`, `ECX`, `EDX`, `ESI` и `EDI`; отметим, что все параметры системных вызовов являются четырёхбайтными значениями — либо целочисленными, либо адресными. Результат выполнения вызова возвращается через регистр `EAX`, причём значение, заключённое между `fffff000h` и `ffffffffh`, свидетельствует о происшедшей ошибке (и представляет собой условный код этой ошибки).

Рассмотрим для примера системный вызов `write`, позволяющий произвести вывод данных через один из открытых потоков ввода-вывода, в том числе запись в открытый файл, а также печать на стандартный вывод (в просторечии «на экран»). Этот системный вызов имеет номер `4` и принимает три параметра: дескриптор (номер) потока ввода-вывода, адрес памяти, где расположены данные, подлежащие выводу, и количе-

ство этих данных в байтах. Отметим, что поток стандартного вывода в ОС Unix имеет дескриптор 1[6]. Таким образом, если мы хотим вывести строку «на экран», то есть сделать то, что делает макрос `PRINT`, нам нужно будет занести число `4` в `EAX`, занести число `1` в `EBX`, занести адрес строки в `ECX` и длину строки — в `EDX`, а затем дать команду `int 80h`, чтобы инициировать программное прерывание.

Другой важный системный вызов — это вызов `_exit`, используемый для завершения программы. Он имеет номер `1` и принимает один параметр, представляющий собой ***код завершения***. Программы используют код завершения, чтобы сообщить операционной системе, успешно ли они справились с возложенной на них задачей: если всё прошло как ожидалось, используется код `0`, если же в ходе работы возникли те или иные ошибки, используются коды `1`, `2` и т. д.

Зная всё это, мы можем написать программу, печатающую строку и сразу после этого завершающуюся; файл `stud_io.inc` и его макросы нам для этого больше не нужны:

```
section .data
msg db  "Hello world", 10
msg_len equ $-msg

section .text
global  _start
_start:
        mov     eax, 4          ; вызов write
        mov     ebx, 1          ; стандартный вывод
        mov     ecx, msg
        mov     edx, msg_len
        int     80h

        mov     eax, 1          ; вызов _exit
        mov     ebx, 0          ; код "успех"
        int     80h
```

## § 4.3.2. Конвенция ОС FreeBSD

Описание конвенции ОС FreeBSD несколько сложнее. Эта система также использует прерывание `80h` и принимает номер системного вызова через регистр `EAX`, но все параметры вызова передаются не через

[6] Точнее было бы сказать, что поток вывода под номером `1` считается стандартным выводом.

регистры, а через стек, подобно тому, как передаются параметры в подпрограммы в соответствии с соглашениями языка Си, то есть в обратном порядке (см. §2.6.7). Как и в ОС Linux, все параметры вызовов представляют собой четырёхбайтные значения. Результат выполнения системного вызова возвращается через регистр `EAX`.

Необходимо отметить ещё одну особенность. Ядро FreeBSD предполагает, что управление ему передано путём обращения к процедуре следующего вида:

```
kernel:
        int 80h
        ret
```

т. е. на момент вызова прерывания в стеке должны находиться параметры вызова и адрес возврата из этой процедуры. Это сделано для удобства написания переносимых программ.

При работе на языке ассемблера выделять вызов прерывания в отдельную подпрограмму не обязательно, достаточно перед командой `int` занести в стек дополнительное «двойное слово», например, выполнив лишний раз команду `push eax` (или любой другой 32-битный регистр). Естественно, после выполнения системного вызова и возврата из него необходимо убрать из стека всё, что туда было занесено; делается это, как и при вызове обычных подпрограмм, путём увеличения регистра `ESP` на нужную величину простой командой `add`.

Описывая в предыдущем параграфе конвенцию ОС Linux, мы для иллюстрации использовали вызовы `write` и `_exit` (см. стр. 49). Аналогичная программа для FreeBSD будет выглядеть следующим образом:

```
section .data
msg db "Hello world", 10
msg_len equ $-msg

section .text
global  _start
_start:
        push    dword msg_len
        push    dword msg
        push    dword 1          ; стандартный вывод
        mov     eax, 4           ; write
        push    eax              ; что угодно
```

```
int     80h
add     esp, 16         ; 4 двойных слова

push    dword 0         ; код "успех"
mov     eax, 1          ; вызов _exit
push    eax             ; что угодно
int     80h
```

Мы не стали очищать стек после системного вызова `_exit`, поскольку он всё равно не возвращает управление.

### § 4.3.3. Некоторые системные вызовы Unix

В вышеприведённых примерах мы рассмотрели системные вызовы `_exit` и `write`; напомним, что `_exit` имеет[7] номер `1` и принимает один параметр — код завершения, а вызов `write` имеет номер `4` и принимает три параметра, а именно номер дескриптора потока вывода (`1` для потока стандартного вывода), адрес области памяти, где расположены выводимые данные, и количество этих данных.

Для ввода данных (как из файлов, так и из стандартного потока ввода, т. е. «с клавиатуры») используется вызов `read`, имеющий номер `3`. Его параметры аналогичны вызову `write`: первый параметр — номер дескриптора потока ввода (для стандартного ввода используется дескриптор `0`), второй параметр — адрес области памяти, в которой необходимо разместить прочитанные данные, а третий — количество байтов, которое надлежит попытаться прочитать. Естественно, область памяти, адрес которой мы передаём вторым параметром, должна иметь размер не менее числа, передаваемого третьим параметром. **Очень важно проанализировать значение, возвращаемое вызовом `read`!** (напомним, что это значение сразу после вызова содержится в регистре `EAX`.) Если чтение прошло успешно, вызов вернёт строго положительное число — количество прочитанных байтов, которое, естественно, не может превышать «заказанное» через третий параметр количество, но вполне может оказаться меньше (например, мы потребовали прочитать 200 байтов, а реально было прочитано только 15). Очень важен случай, когда `read` возвращает число `0` — это свидетельствует о том, что в используемом потоке ввода возникла ситуация «конец файла». При чтении из файлов это значит, что весь файл прочитан и больше в нём

[7] Во всяком случае, в системах GNU/Linux и FreeBSD; в дальнейшем, если нет явных указаний, подразумевается, что сказанное верно как минимум для этих двух систем.

данных нет. Однако «конец файла» может произойти не только при чтении из настоящего файла; так, при вводе с клавиатуры в ОС Unix можно сымитировать ситуацию «конец файла», нажав комбинацию клавиш Ctrl-D.

**Помните, что программа, в которой используется вызов `read` и не производится анализ его результата, заведомо не может быть правильной.** Действительно, мы в этом случае не можем знать, сколько первых байтов нашей области памяти содержат реально прочитанные данные, а сколько оставшихся продолжают содержать произвольный «мусор» — а значит, какая-либо осмысленная работа с этими данными невозможна.

При чтении, как и при использовании других системных вызовов, может произойти ошибка. В ОС Linux это легко обнаружить по отрицательному значению регистра `EAX` после возврата из вызова; в ОС FreeBSD для указания на то, что произошла ошибка, системные вызовы используют флаг `CF` (carry flag): если вызов завершился успешно, на выходе из него этот флаг будет сброшен, если же произошла ошибка, то флаг будет установлен. Это касается и вызова `read`, и рассмотренного ранее вызова `write` (мы не обрабатывали ошибочные ситуации, чтобы не усложнять наши примеры, но это не значит, что ошибки не могут произойти), и всех остальных системных вызовов.

На момент запуска программы для неё, как правило, открыты потоки ввода-вывода с номерами `0` (стандартный ввод), `1` (стандартный вывод) и `2` (поток для выдачи сообщений об ошибках), так что мы можем применять вызов `read` к дескриптору `0`, а к дескрипторам `1` и `2` — вызов `write`. Часто, однако, задача требует создания иных потоков ввода-вывода, например, для чтения и записи файлов на диске. Прежде чем мы сможем работать с файлом, его необходимо *открыть*, в результате чего у нас появится ещё один поток ввода-вывода со своим номером (дескриптором). Делается это с помощью системного вызова `open`, имеющего номер `5`. Вызов принимает три параметра. Первый параметр — адрес строки текста, задающей имя файла; имя должно заканчиваться нулевым байтом, который служит в качестве ограничителя. Второй параметр — число, задающее режим использования файла (чтение, запись и пр.); значение этого параметра формируется как битовая строка, в которой каждый бит означает определённую особенность режима, например, доступность только на запись, разрешение создать новый файл, если его нет, и т. п. К сожалению, расположение этих битов различно для ОС Linux и ОС FreeBSD; некоторые из флагов вместе с их описаниями и численными значениями приведены в таблице 4.1. Отметим, что

| название | описание | значение для | |
|---|---|---|---|
| | | Linux | FreeBSD |
| O_RDONLY | только чтение | 000h | 000h |
| O_WRONLY | только запись | 001h | 001h |
| O_RDWR | чтение и запись | 002h | 002h |
| O_CREAT | разрешить создание файла | 040h | 200h |
| O_EXCL | потребовать создание файла | 080h | 800h |
| O_TRUNC | если файл существует, уничтожить его содержимое | 200h | 400h |
| O_APPEND | если файл существует, дописывать в конец | 400h | 008h |

Таблица 4.1. Некоторые флаги для второго параметра вызова `open`

наиболее часто встречаются два варианта для этого параметра. Первый из них — открытие файла только для чтения, в обеих рассматриваемых системах этот случай задаётся числом `0`. Второй случай — открытие файла на запись, при котором файл создаётся, если его не было, а если он был, то его старое содержимое теряется (в программах на Си это задаётся комбинацией `O_WRONLY|O_CREAT|O_TRUNC`). Для Linux соответствующее числовое значение — `241h`, для FreeBSD — `601h`.

Третий параметр вызова `open` используется только в случае создания файла и задаёт *права доступа* для него. Подробное описание этого параметра мы опускаем, отметим только, что в большинстве случаев его следует задать равным восьмеричному числу `0666q`.

Для вызова `open` особенно важен анализ его возвращаемого значения и проверка, не произошла ли ошибка. Вызов `open` может завершиться с ошибкой в силу массы причин, большинство из которых программист никак не может ни предотвратить, ни предсказать: например, кто-то может неожиданно стереть файл, который мы собирались открыть на чтение, или запретить нам доступ к директории, где мы намеревались создать новый файл. Итак, после выполнения вызова `open` нам необходимо проверить, не содержит ли регистр `EAX` отрицательное значение (в ОС Linux) или не взведён ли флаг `CF` (в ОС FreeBSD).

Если вызов закончился успешно, то регистр `EAX` содержит ***дескриптор открытого файла*** (потока ввода или вывода). Именно этот дескриптор теперь следует использовать в качестве первого параметра в вызовах `read` и `write` для работы с файлом. Как правило, это значение

следует сразу же после вызова скопировать в специально отведённую для него область памяти.

Когда все действия с файлом завершены, его следует закрыть. Это делается с помощью вызова `close`, имеющего номер 6. Вызов принимает один параметр, равный дескриптору закрываемого файла. После этого поток ввода-вывода с таким дескриптором перестаёт существовать; последующие вызовы `open` могут снова использовать тот же номер дескриптора.

Задача в ОС Unix может узнать свой номер (так называемый идентификатор процесса) с помощью вызова `getpid`, а также номер своего непосредственного «предка» (процесса, создавшего данный процесс) с помощью вызова `getppid`. Вызов `getpid` в обеих рассматриваемых системах имеет номер 20, тогда как вызов `getppid` имеет номер 64 в ОС Linux и номер 39 в ОС FreeBSD. Оба вызова не принимают параметров; запрашиваемый номер возвращается в качестве результата работы вызова через регистр `EAX`. Отметим, что эти два вызова всегда завершаются успешно, ошибкам тут просто неоткуда взяться.

Системный вызов `kill` (номер 37) позволяет отправить сигнал процессу с заданным номером. Вызов принимает два параметра, первый задаёт номер процесса[8], второй задаёт номер сигнала; в частности, сигнал № 15 (`SIGTERM`) предписывает процессу завершиться (но процесс может этот сигнал перехватить и завершиться не сразу, либо вообще не завершаться), а сигнал № 9 (`SIGKILL`) уничтожает процесс, причём этот сигнал нельзя ни перехватить, ни игнорировать.

Ядра операционных систем семейства Unix поддерживают сотни разнообразных системных вызовов; заинтересованные читатели могут найти информацию об этих вызовах в сети Интернет или в специальной литературе. Отметим, что для ознакомления с информацией о системных вызовах желательно знать язык программирования Си, да и работа на уровне системных вызовов с помощью языка Си строится гораздо проще. Более того, некоторые системные вызовы в отдельных системах могут не поддерживаться ядром, а вместо этого эмулироваться библиотечными функциями Си, что делает их использование в программах на языке ассемблера практически невозможным.

В этой связи нелишним будет напомнить, что язык ассемблера мы рассматриваем с учебной, а не практической целью. Программы, предназначенные для практического применения, лучше писать на Си или на других подходящих языках высокого уровня.

[8] На самом деле можно отправить сигнал сразу группе процессов или даже всем процессам в системе, но подробное описание этого выходит за рамки нашего курса.

## § 4.4. Параметры командной строки

При работе в операционной среде ОС Unix мы, как правило, запускаем программы, указывая кроме их имён ещё и определённые параметры — имена файлов, опции и т. п. Так, при запуске ассемблера NASM мы можем написать что-то вроде

```
nasm -f elf prog.asm
```

Слова, указанные после имени программы, называются ***параметрами командной строки***. В данном случае этих аргументов три: ключ «`-f`», слово «`elf`», обозначающее нужный нам формат результата трансляции, и имя файла «`prog.asm`». Отметим, что и само имя программы, в данном случае «`nasm`», считается элементом командной строки. Иначе говоря, командная строка представляет собой массив строк, состоящий в данном случае из четырёх элементов: «`nasm`», «`-f`», «`elf`» и «`prog.asm`».

Естественно, мы и сами можем написать программу, получающую те или иные сведения через командную строку. При запуске программы операционная система отводит в её адресном пространстве специальную область памяти, в которой располагает строки, составляющие командную строку. Информация об адресах этих строк вместе с их общим количеством для удобства помещается в стек запускаемой задачи, после чего управление передаётся нашей программе. Таким образом, в тот момент, когда наша программа начинает выполняться с метки `_start`, на вершине стека (то есть по адресу `[esp]`) располагается четырёхбайтное целое число, равное количеству элементов командной строки (включая имя программы), в следующей позиции стека (по адресу `[esp+4]`) располагается адрес в памяти, где находится имя, по которому нашу программу вызвали, далее (по адресу `[esp+8]`) находится адрес первого параметра, потом второго параметра и т. д. Каждый элемент командной строки хранится в памяти в виде строки (массива символов), ограниченной справа нулевым байтом.

Для примера рассмотрим программу, печатающую параметры своей командной строки (включая нулевой). Пользоваться средствами `stud_io.inc` мы уже не станем, поскольку знаем, как без них обойтись. Для использования вызова `write` нам понадобится знать длину каждой печатаемой строки, поэтому для удобства мы опишем подпрограмму `strlen`, получающую в качестве параметра через стек адрес строки и возвращающую через регистр `EAX` длину этой строки (предполагается, что конец строки обозначен нулевым байтом). Кроме того, отдельную

подпрограмму (`newline`) опишем для печати символа перевода строки; при этом нам потребуется область памяти из одного байта, равного `10`, то есть коду перевода строки, чтобы передавать её адрес вызову `write`, и мы эту область памяти отведём прямо в секции `.text`[9] вместе с кодом подпрограммы `newline` сразу после команды `ret`, пометив локальной меткой.

Ещё один своеобразный момент состоит в том, что наша программа будет расчитана как для работы с ОС Linux, так и для работы с ОС FreeBSD. Поскольку системные вызовы в этих ОС выполняются по-разному, мы воспользуемся директивами условной компиляции для выбора того или иного текста. Эти директивы будут предполагать, что при компиляции под ОС Linux мы определяем (в командной строке NASM) макросимвол `OS_LINUX`, а при работе под FreeBSD — символ `OS_FREEBSD`. Таким образом, при работе под ОС Linux наш пример (назовём его `cmdl.asm`) нужно будет компилировать с помощью команды

```
nasm -f elf -dOS_LINUX cmdl.asm
```

а при работе под ОС FreeBSD — командой

```
nasm -f elf -dOS_FREEBSD cmdl.asm
```

Итак, пишем текст:

```
section .text
global  _start

strlen:         ; arg1 == address of the string
        push ebp
        mov ebp, esp
        push esi
        xor eax, eax
        mov esi, [ebp+8]   ; arg1
.lp:    cmp byte [esi], 0
        jz .quit
        inc esi
        inc eax
        jmp short .lp
.quit:  pop esi
        pop ebp
```

[9] Мы можем так поступить, поскольку эту область памяти наша программа не меняет; если бы это было не так, пришлось бы располагать её в секции `.data`

```
        ret

newline:
        pushad
%ifdef OS_FREEBSD
        push dword 1
        push dword .nwl
        push dword 1 ; stdout
        mov eax, 4 ; write
        push eax
        int 80h
        add esp, 16
%elifdef OS_LINUX
        mov edx, 1
        mov ecx, .nwl
        mov ebx, 1
        mov eax, 4
        int 80h
%else
%error please define either OS_FREEBSD or OS_LINUX
%endif
        popad
        ret
.nwl    db 10

_start:
        mov ecx, [esp]
        mov esi, esp
        add esi, 4
again:  push dword [esi]
        call strlen
        add esp, 4
        push esi
        push ecx
%ifdef OS_FREEBSD
        push eax
        push dword [esi]
        push dword 1 ; stdout
        mov eax, 4 ; write
        push eax
```

```
        int 80h
        add esp, 16
%else
        mov edx, eax
        mov ecx, [esi]
        mov ebx, 1
        mov eax, 4
        int 80h
%endif
        call newline
        pop ecx
        pop esi
        add esi, 4
        loop again

%ifdef OS_FREEBSD
        push dword 0
        mov eax, 1 ; _exit
        push eax
        int 80h
%else
        mov ebx, 0
        mov eax, 1
        int 80h
%endif
```

## § 4.5. Пример: копирование файла

Рассмотрим ещё один пример программы, активно взаимодействующей с операционной системой. Эта программа будет получать через параметры командной строки имена двух файлов — оригинала и копии и создавать копию под заданным именем с заданного оригинала. Наша программа будет работать достаточно просто: проверив, что ей действительно передано два параметра, она попытается открыть первый файл на чтение, второй файл — на запись и, если ей это удалось, то циклически читать из первого файла данные порциями по 4096 байт, пока не возникнет ситуация «конец файла». Сразу после чтения каждой порции программа будет записывать прочитанное во второй файл. Настоящая

команда `cp`, предназначенная для копирования файлов, устроена гораздо сложнее, но для нашего учебного примера лишняя сложность не нужна.

Ясно, что нашей программе предстоит активно пользоваться системными вызовами. Дело осложняется тем, что нам хотелось бы, конечно, написать программу, которая будет успешно компилироваться и работать как под ОС Linux, так и под ОС FreeBSD. Как мы видели на примере программы из предыдущего параграфа, это требует довольно громоздкого оформления каждого системного вызова директивами условной компиляции. Предыдущий пример, содержащий всего три системных вызова, можно было написать, не особенно задумываясь над этой проблемой, что мы и сделали; иное дело — программа, в которой предполагается больше десятка обращений к операционной системе.

Чтобы не допустить загромождения нашего исходного кода однообразными, но при этом объёмными (и, значит, отвлекающими внимание) конструкциями, мы напишем один многострочный макрос, который и будет осуществлять системный вызов (точнее, он будет генерировать ассемблерный код для осуществления системного вызова). В тексте этого макроса и будут заключены все различия в организации системных вызовов для Linux и FreeBSD.

Макрос будет принимать на вход произвольное количество параметров, не меньшее одного; первый параметр будет задавать номер системного вызова, остальные — значения параметров системного вызова. Отметим, что для ОС Linux наш макрос откажется работать с более чем пятью параметрами, поскольку они уже не уместятся в регистры; для FreeBSD такого ограничения нет.

При передаче параметров в макрос и раскладывании их по соответствующим регистрам (в варианте для Linux) мы применим приём, который уже встречали (см. комментарий на стр. 28) — занесение всех параметров в стек с последующим их извлечением в нужные регистры. В варианте для FreeBSD никакого раскладывания по регистрам нам не требуется, зато требуется занести параметры в стек уже для использования их самим системным вызовом. Таким образом, в обоих случаях тело макроса можно начать с занесения в стек всех его параметров (в обратном порядке, чтобы не пришлось их как-либо переупорядочивать в варианте для FreeBSD). Для этого мы воспользуемся директивой `%rotate` точно так же, как мы это уже делали при написании макроса `pcall` (см. стр. 30).

После этого в варианте для FreeBSD достаточно занести номер вызова в `EAX`, и можно инициировать прерывание; в варианте для Linux

всё не так просто, нужно ещё извлечь из стека параметры и расположить их в регистрах, причём для различного количества параметров будут задействоваться различные наборы регистров; чтобы корректно обработать всё это, нам придётся написать целый ряд вложенных друг в друга директив условной компиляции, срабатывающих в зависимости от количества переданных макросу параметров.

После возврата из системного вызова наши действия также различаются в зависимости от используемой операционной системы. В случае ОС Linux результат вызова находится в регистре `EAX`, отрицательное значение указывает на возникшую ошибку, в стеке ничего лишнего нет. В случае ОС FreeBSD на ошибку указывает взведённый флаг `CF`, в регистре `EAX` может находиться как результат, так и код ошибки, а в стеке всё ещё лежат параметры вызова, так что стек нуждается в очистке. Мы поступим следующим образом: в случае ОС Linux оставим всё как есть, в случае же ОС FreeBSD, во-первых, проверим флаг `CF`, и если он взведён, изменим знак регистра `EAX` на противоположный с помощью команды `neg`. Таким образом, на выходе мы, как и для ОС Linux, будем иметь в `EAX` неотрицательное значение в случае успеха и отрицательное — в случае ошибки; после этого мы совершенно спокойно можем испортить содержимое регистра флагов, что, кстати, и произойдёт на следующей команде — мы очистим стек от ненужных уже параметров обычной командой `add`, которая, как известно, выставляет флаги (включая `CF`) уже в соответствии со своим результатом.

Окончательно наш макрос будет выглядеть так:

```
%macro        syscall 1-*
%rep %0
%rotate -1
        push dword %1
%endrep
%ifdef OS_FREEBSD
        mov eax, [esp]
        int 80h
        jnc %%sc_ok
        neg eax
%%sc_ok:
        add esp, (%0-1)*4
%elifdef OS_LINUX
        pop eax
 %if %0 > 1
        pop ebx
```

```
    %if %0 > 2
          pop ecx
     %if %0 > 3
          pop edx
      %if %0 > 4
          pop esi
       %if %0 > 5
          pop edi
        %if %0 > 6
         %error "Too many params for Linux syscall"
        %endif
       %endif
      %endif
     %endif
    %endif
   %endif
          int 80h
%else
%error Please define either OS_LINUX or OS_FREEBSD
%endif
%endmacro
```

Текст макроса, конечно, получился достаточно длинным, но это компенсируется сокращением объёма основного кода. Так, рассказывая о конвенциях системных вызовов, мы привели код программы, печатающей одну строку, в варианте для Linux (стр. 50) и FreeBSD (стр. 51). С использованием вышеприведённого макроса мы можем написать так:

```
section .data
msg db  "Hello world", 10
msg_len equ $-msg
section .text
global  _start
_start:
        syscall 4, 1, msg, msg_len
        syscall 1, 0
```

и всё, причём эта программа будет компилироваться и правильно работать под обеими системами, нужно только не забывать указывать NASM'у флаг `-dOS_LINUX` или `-dOS_FREEBSD`.

Вернёмся к нашей задаче копирования. В программе нам потребуется буфер для временного хранения данных, в который мы будем считывать очередную порцию данных из первого файла, чтобы затем запи-

сать её во второй файл. Кроме того, нам будут нужны переменные для хранения дескрипторов файлов (хранить их в регистрах будет сложно, ведь каждый системный вызов может испортить значения регистров); соответствующие переменные мы назовём `fdsrc` и `fddest`. Наконец, мы для удобства заведём переменные для хранения количества параметров командной строки и адреса начала массива указателей на параметры командной строки, назвав эти переменные `argc` и `argvp`. Все эти переменные не требуют начальных значений и могут, таким образом, быть расположены в секции `.bss`:

```
section .bss
buffer  resb    4096
bufsize equ     $-buffer
fdsrc   resd    1
fddest  resd    1
argc    resd    1
argvp   resd    1
```

Наша программа может обнаружить одну из трёх ошибок: пользователь может указать неправильное количество параметров командной строки, может указать несуществующий или недоступный файл в качестве источника данных, либо может указать в качестве целевого такой файл, который мы по каким-то причинам не сможем открыть на запись. В первом случае пользователю следует объяснить, с какими параметрами следует запускать нашу программу, в остальных двух — просто сообщить о происшедшей ошибке. Все три сообщения об ошибках мы расположим в секции `.data` в виде инициализированных переменных:

```
section .data
helpmsg db 'Usage: copy <src> <dest>', 10
helplen equ $-helpmsg
err1msg db "Couldn't open source file for reading", 10
err1len equ $-err1msg
err2msg db "Couldn't open destination file for writing", 10
err2len equ $-err1msg
```

Теперь мы можем приступить к написанию секции `.text`, то есть самой программы, и в самом начале мы проверим, что нам передано ровно два параметра. Для этого мы извлечём из стека лежащее на его вершине число, обозначающее количество элементов командной строки, занесём его в переменную `argc`. Заодно на всякий случай сохраним адрес текущей вершины стека в переменной `argvp`, но извлекать из стека больше

ничего не будем, так что в области стека у нас окажется массив адресов строк–элементов командной строки. Проверим, что в переменной `argc` оказалось число 3; правильная командная строка должна в нашем случае состоять из трёх элементов: имени самой программы и двух параметров. В случае, если количество параметров окажется неверным, напечатаем пользователю сообщение об ошибке и выйдем:

```
section .text
global  _start
_start:
        pop dword [argc]
        mov [argvp], esp
        cmp dword [argc], 3
        je .args_count_ok
        syscall 4, 2, helpmsg, helplen
        syscall 1, 1
.args_count_ok:
```

Следующим нашим действием должно стать открытие файла, имя которого задано первым параметром командной строки, на чтение. Мы помним, что в переменной `argvp` находится адрес в памяти (стековой), начиная с которого располагаются адреса элементов командной строки. Извлечём адрес из `argvp` в регистр `ESI`, затем возьмём четырёхбайтное значение по адресу `[esi+4]` — это и будет адрес первого параметра командной строки, то есть строки, задающей имя файла, который надо читать и копировать. Для хранения адреса воспользуемся регистром `EDI`, после чего сделаем вызов `open`. Нам придётся использовать два параметра — собственно адрес имени файла и режим его использования, который будет в данном случае равен `0` (`O_RDONLY`). Результат работы системного вызова нам обязательно надо будет проверить; напомним, что наш макрос `syscall` устроен так, чтобы отрицательное значение `EAX` указывало на ошибку, а неотрицательное — на успешное выполнение вызова; в применении к вызову `open` результатом успешного его выполнения является дескриптор нового потока ввода-вывода, в данном случае это поток ввода, связанный с копируемым файлом. В случае успеха сохраним полученный дескриптор в переменной `fdsrc`, в случае неудачи — выдадим сообщение об ошибке и выйдем.

```
        mov esi, [argvp]
        mov edi, [esi+4]
        syscall 5, edi, 0 ; O_RDONLY
        cmp eax, 0
```

```
        jge .source_open_ok
        syscall 4, 2, err1msg, err1len
        syscall 1, 2
.source_open_ok:
        mov [fdsrc], eax
```

Настало время открыть второй файл на запись. Для извлечения его имени из памяти воспользуемся точно так же регистрами `ESI` и `EDI`, после чего выполним системный вызов `open`, в случае ошибки выдадим сообщение и выйдем, в случае успеха сохраним дескриптор в переменной `fddest`. Вызов `open` в этот раз будет несколько сложнее. Во-первых, режим открытия на этот раз задаётся флажками `O_WRONLY`, `O_CREAT` и `O_TRUNC`, два из которых, как это обсуждалось на стр. 54, имеют различные числовые значения в ОС Linux и ОС FreeBSD. Во-вторых, поскольку в этот раз возможно создание нового файла, наш системный вызов должен получить ещё и третий параметр, который, как мы ранее отмечали, обычно равен `666o`. С учётом всего этого получится такой код:

```
        mov esi, [argvp]
        mov edi, [esi+8]
%ifdef OS_LINUX
        syscall 5, edi, 241h, 0666o
%else           ; assume it's FreeBSD
        syscall 5, edi, 601h, 0666o
%endif
        cmp eax, 0
        jge .dest_open_ok
        syscall 4, 2, err2msg, err2len
        syscall 1, 3
.dest_open_ok:
        mov [fddest], eax
```

Наконец, напишем основной цикл. В нём мы будем выполнять чтение из первого файла, анализировать его результат, и если достигнут конец файла (в `EAX` значение `0`) или произошла ошибка (отрицательное значение), то будем выходить из цикла, ну а если чтение прошло успешно, то нужно будет записать всё прочитанное (то есть столько байтов из области память `buffer`, какое число содержится в `EAX`) во второй файл.

```
.again: syscall 3, [fdsrc], buffer, bufsize
        cmp eax, 0
```

```
        jle .end_of_file
        syscall 4, [fddest], buffer, eax
        jmp .again
```

Выход из цикла мы производили переходом на метку `end_of_file`; рано или поздно наша программа, достигнув конца первого файла, перейдёт на эту метку, после чего нам останется только закрыть оба файла вызовом `close` и завершить программу:

```
.end_of_file:
        syscall 6, [fdsrc]
        syscall 6, [fddest]
        syscall 1, 0
```

Отметим, что все метки в основной программе, кроме метки `_start`, мы сделали локальными (их имена начинаются с точки). Так делать не обязательно, но такой подход к меткам (все метки, к которым не предполагается обращаться откуда-то издалека, делать локальными) позволяет в более крупных программах избежать проблем с конфликтами имён.

# Глава 5. Модульное программирование

## § 5.1. Что такое модули и зачем они нужны

До сих пор все программы, которые мы писали на языке ассемблера, умещались в одном файле. Иногда мы использовали несколько файлов, но соединение их воедино производилось на этапе макропроцессирования, то есть ещё до начала перевода программы в машинный код.

Пока исходный текст программы состоит из нескольких десятков строк, его действительно удобнее всего хранить в одном файле. С увеличением объема программы, однако, работать с одним файлом становится всё труднее и труднее, и тому можно назвать несколько причин. Во-первых, длинный файл элементарно тяжело перелистывать. Во-вторых, как правило, программист в каждый момент времени работает только с небольшим фрагментом исходного кода, старательно выкидывая из головы остальные части программы, чтобы не отвлекаться, и в этом плане было бы лучше, чтобы фрагменты, не находящиеся в работе в настоящий момент, располагались бы где-нибудь подальше, то есть так, чтобы не попадаться на глаза программисту даже случайно. В-третьих, если программа разбита на отдельные файлы, в ней оказывается гораздо проще найти нужное место, подобно тому, как проще найти нужную бумагу в шкафу с офисными папками, нежели в большом ящике безо всяких папок. Наконец, часто бывает так, что один и тот же фрагмент кода используется в разных программах — а ведь его, скорее всего, так или иначе приходится время от времени редактировать, чтобы, например, исправить ошибки, и тут уже совершенно очевидно, что гораздо проще исправить файл в одном месте и скопировать (файл целиком)

во все остальные проекты, чем исправлять один и тот же фрагмент, который вставлен в разные файлы.

Разбивка текста программы на файлы, соединяемые директивами `%include` или их аналогами, снимает часть проблем, но, к сожалению, не все, поскольку такой набор файлов остаётся, как говорят программисты, *одной единицей трансляции* — иначе говоря, мы можем их транслировать с помощью ассемблера (или с помощью компилятора, если мы пишем на языке высокого уровня) только все вместе, за один приём. Прежде всего тут возникает проблема со скоростью трансляции. Современные компиляторы и ассемблеры работают довольно быстро, но объемы наиболее серьёзных программ таковы, что их полная перекомпиляция может занять несколько часов, а иногда и несколько суток. Если после внесения любого, даже самого незначительного изменения в программу нам, чтобы посмотреть, что получилось, придётся ждать сутки (да и пару часов — этого уже будет достаточно) — работать станет совершенно невозможно. Более того, программисты практически всегда используют так называемые ***библиотеки*** — комплекты готовых подпрограмм, которые почти никогда не изменяются и, соответственно, постоянно тратить время на их перекомпиляцию было бы несколько глупо. Наконец, проблемы создают и постоянно возникающие конфликты имён: чем больше объём кода, тем больше в нём требуется меток и других идентификаторов, растёт вероятность случайных совпадений, а сделать с этим при трансляции в один приём почти ничего нельзя — ведь даже локальные метки, как мы уже говорили, на самом деле представляют собой не более чем укороченную запись более длинных глобальных меток.

Все эти проблемы позволяет решить техника ***раздельной компиляции***. Суть её в том, что программа создаётся в виде множества обособленных частей, каждая из которых транслируется отдельно. Такие части называются ***единицами трансляции*** или ***модулями***. Чаще всего в роли модулей выступают отдельные файлы. Обычно в виде обособленной единицы трансляции оформляют набор логически связанных между собой подпрограмм; в модуль также помещают и всё необходимое для их работы — например, глобальные переменные, если такие есть, а также всевозможные константы и прочее. Каждый модуль транслируется отдельно; в результате трансляции каждого из них получается ***объектный файл***, обычно имеющий суффикс `.o`. Затем с помощью системного редактора связей из набора объектных файлов получают исполняемый файл.

Очень важным свойством модуля является наличие у него собственного ***пространства имён***. Метки, введённые в модуле, будут видны только из других мест того же модуля, если только мы специально не объявим их «глобальными» (напомним, что в языке ассемблера NASM это делается директивой `global`). Часто бывает так, что модуль вводит несколько десятков, а иногда и сотен меток, но все они оказываются нужны только в нём самом, а из всей остальной программы требуются обращения лишь к одной-двум процедурам. Это практически снимает проблему конфликтов имён: в разных модулях могут появляться метки с одинаковыми именами, и это никак нам не мешает, если только они не глобальные. Технически это означает, что при трансляции исходного текста модуля в объектный код все метки, кроме объявленных глобальными, исчезают, так что в объектном файле содержится уже только информация об именах глобальных меток.

Интересно, что собственные пространства имён модулей позволяют решить не только проблему конфликта имён, но и проблему простейшей «защиты от дурака», особенно актуальной в крупных программных разработках, в которых принимает участие несколько человек. Если автор модуля не предполагает, что та или иная процедура будет вызываться из других модулей, либо что переменная не должна изменяться никак иначе, чем процедурами того же модуля, то ему достаточно не объявлять соответствующие метки глобальными, и можно ни о чём не беспокоиться — обратиться к ним другие программисты не смогут чисто технически. Такое сокрытие деталей реализации той или иной подсистемы в программе называется ***инкапсуляцией*** и позволяет, например, более смело исправлять код модулей, не боясь, что другие модули при этом перестанут работать: достаточно сохранять неизменными и работающими глобальные метки.

## § 5.2. Поддержка модулей в NASM

Ассемблер NASM, естественно, поддерживает модульное программирование, вводя для этого два основных понятия: ***глобальные метки*** и ***внешние метки***. С первыми из них мы уже знакомы: такие метки объявляются директивой `global` и, как мы уже знаем, отличаются от обычных тем, что информация о них включается в объектный файл модуля и становится, таким образом, видна системному редактору связей.

Что касается внешних меток, то это, напротив, метки, ***введения которых мы ожидаем от других модулей***. Чаще всего это просто имя подпрограммы (реже — глобальной переменной), которая описана где-то в другом модуле, но к которой нам необходимо обратиться.

Чтобы это стало возможным, необходимо сообщить ассемблеру о существовании этой метки. Действительно, ассемблер во время трансляции видит только текст одного модуля и ничего не знает о том, что в других модулях объявлены те или иные метки, так что, если мы попытаемся обратиться к метке из другого модуля, никак не сообщив ассемблеру о факте её существования, мы попросту получим сообщение об ошибке.

Для этого ассемблер NASM вводит директиву `extern`. Например, если мы пишем модуль, в котором хотим обратиться к процедуре `myproc`, а сама эта процедура описана где-то в другом месте, то, чтобы сообщить об этом, следует написать:

```
extern myproc
```

Такая строка приказывает ассемблеру буквально следующее: «метка `myproc` существует, хотя её и нет в текущем модуле, так что, встретив такую метку, просто сгенерируй соответствующий объектный код, а конкретный адрес вместо этой метки потом подставит редактор связей».

## § 5.3. Пример

В качестве примера многомодульной программы мы напишем простую программу, которая спрашивает у пользователя его имя, а затем здоровается с ним по имени. Работу со строками мы на этот раз организуем так, как это обычно делается в программах на языке Си: будем использовать нулевой байт в качестве признака конца строки. Головная программа будет зависеть от двух основных подпрограмм, `putstr` и `getstr`, каждую из которых мы вынесем в отдельный модуль. Подпрограмме `putstr` потребуется посчитать длину строки, чтобы напечатать всю строку за одно обращение к операционной системе; для такого подсчёта мы используем функцию `strlen`, уже знакомую нам по программе из § 4.4. Её мы тоже вынесем в отдельный модуль. Наконец, организацию вызова `_exit` мы тоже вынесем в подпрограмму (назовём её `quit`) и в отдельный модуль. Все модули назовём так же, как и вынесенные в них подпрограммы: `putstr.asm`, `getstr.asm`, `strlen.asm` и `quit.asm`.

Для организации системных вызовов мы используем макрос `syscall`, который мы описали на стр. 61. Его мы также вынесем в отдельный файл, но полноценным модулем этот файл быть не сможет. Действительно, модуль — это единица трансляции, тогда как макрос, вообще говоря, не может быть ни во что оттранслирован: как мы отме-

чали ранее, в ходе трансляции макросы полностью исчезают и в объектном коде от них ничего не остаётся. Это и понятно, ведь макросы представляют собой набор указаний не для процессора, а для самого ассемблера, и чтобы от макроса была какая-то польза, ассемблер должен, разумеется, видеть определение макроса в том месте, где он встретит обращение к этому макросу. Поэтому файл, содержащий наш макрос `syscall`, мы будем подсоединять к другим файлам с помощью директивы `%include` на стадии препроцессирования (в отличие от модулей, которые собираются в единое целое существенно позже — после завершения трансляции, с помощью редактора связей). Этот файл мы назовём `syscall.inc`; с него мы вполне можем начать, открыв его для редактирования и набрав в нём ровно такое определение макроса, какое было дано на стр. 61; ничего другого в этом файле набирать не требуется.

Следующим мы напишем файл `strlen.asm`. Он будет выглядеть так:

```
global  strlen

section .text
; procedure strlen
; [ebp+8] == address of the string
strlen: push ebp
        mov ebp, esp
        xor eax, eax
        mov esi, [ebp+8]
.lp:    cmp byte [esi], 0
        jz .quit
        inc esi
        inc eax
        jmp short .lp
.quit:  pop ebp
        ret
```

Первая строчка файла указывает, что в этом модуле будет определена метка `strlen` и эту метку необходимо сделать видимой из других модулей. Вообще говоря, мы могли бы поставить эту директиву где угодно, но лучше вынести её в начало, чтобы при первом же взгляде на текст модуля можно было догадаться, для чего он нужен. Подробно комментировать текст процедуры мы не будем, поскольку он нам уже знаком.

Имея в своём распоряжении процедуру `strlen`, напишем модуль `putstr.asm`. Процедура `putstr` будет вызывать `strlen` для подсчёта длины строки, а затем обращаться к системному вызову `write`:

```
%include "syscall.inc"     ; нужен макрос syscall

global  putstr             ; модуль описывает putstr
extern  strlen             ; а сам использует strlen

section        .text
; procedire putstr
; [ebp+8] = address of the string
putstr: push ebp           ; стандартное начало
        mov ebp, esp       ;   подпрограммы
        push dword [ebp+8] ; вызываем strlen для
        call strlen        ;   подсчёта длины строки
        add esp, 4         ; результат теперь в EAX
                           ; вызываем write
        syscall 4, 1, [ebp+8], eax
        mov esp, ebp       ; стандартное завершение
        pop ebp            ;   подпрограммы
        ret
```

Теперь настал черёд самого сложного из модулей нашей программы — модуля `getstr`. Процедура `getstr` будет получать на вход адрес буфера, в котором следует разместить прочитанную строку, и (на всякий случай) длину этого буфера, чтобы не допустить его переполнения, если пользователю придёт в голову набрать строку, которая в наш буфер не поместится. Для упрощения реализации мы будем считывать строку по одному символу; конечно, в настоящих программах так не делают, но наша задача сейчас не в том, чтобы получить эффективную программу, так что мы вполне можем немного облегчить себе жизнь. Подпрограмма `getstr` будет использовать локальную переменную, которую в комментариях мы назовём `I` и которая, как и все локальные переменные, будет располагаться в стековом фрейме, для чего мы в начале процедуры соответствующим образом изменим указатель стека. В переменной `I` будет содержаться *текущее количество прочитанных символов*, изначально равное нулю. Далее процедура будет в цикле читать по одному символу с помощью системного вызова `read`. Чтение будет прекращено при наступлении одного из следующих условий: либо `read` вернёт что-либо отличное от `1`, что в данном случае будет означать наступление ситуа-

ции «конец файла» или ошибку; либо код прочитанного символа будет равен 10, то есть это окажется символ перевода строки (этот код генерирует клавиша Enter); либо, наконец, в буфере останется место только под завершающий нулевой байт, что проверяется условием `I+1≥buflen`.

После выхода из цикла а конец буфера записывается ограничительный нулевой байт. В случае, если причиной выхода из цикла был прочитанный код символа перевода строки, нулевой байт записывается на его место, чтобы в буфере никаких переводов строки не содержалось; это достигается уменьшением переменной `I` перед выходом из цикла.

Полностью текст модуля `getstr.asm` будет выглядеть так:

```
%include "syscall.inc"     ; нужен макрос syscall

global getstr              ; модуль описывает getstr

section .text
; procedure getstr
; [ebp+8]  = address of buffer
; [ebp+12] = length of buffer
getstr: push ebp            ; стандартное начало
        mov ebp, esp        ;   подпрограммы
        sub esp, 4          ; место под переменную I
        xor eax, eax        ; eax:=0
        mov [ebp-4], eax    ; I:=0
.again:                     ; начало главного цикла
        mov eax, [ebp+8]    ; заносим адрес в EAX
        add eax, [ebp-4]    ; прибавляем к нему I
                            ; вызываем read
        syscall 3, 0, eax, 1
        cmp eax, 1          ; вернул ли он 1?
        jne .endstr         ; нет - выйти из цикла
        mov eax, [ebp+8]    ; заносим адрес в EAX
        add eax, [ebp-4]    ; прибавляем к нему I
        mov bl, [eax]       ; считанный байт (в BL)
        cmp bl, 10          ; равен 10?
        jne .noeol          ; нет - перепрыгиваем
        dec dword [ebp-4]   ; да - уменьшаем I
        jmp .endstr         ;   и выходим из цикла
.noeol: mov eax, [ebp-4]    ; загружаем I
        inc eax             ; теперь в EAX зн. I+1
        cmp eax, [ebp+12]   ; не превышает ли arg2?
```

```
        jae .endstr         ; да - выходим из цикла
        inc dword [ebp-4]   ; увеличиваем I
        jmp .again          ; продолжаем цикл
.endstr:
        mov eax, [ebp+8]    ; загружаем адрес в EAX
        add eax, [ebp-4]    ; прибавляем I
        inc eax             ; прибавляем 1
        xor bl, bl          ; обнуляем BL
        mov [eax], bl       ; заносим 0 в конец строки
        mov esp, ebp        ; стандартный выход
        pop ebp             ;   из подпрограммы
        ret
```

Напишем теперь самый простой из наших модулей — `quit.asm`:

```
%include "syscall.inc"
global  quit
section .text
quit:   syscall 1, 0
```

Все подпрограммы готовы, и мы можем приступать к написанию головного модуля, который мы назовём `greet.asm`. Поскольку все обращения к системным вызовам мы вынесли в подпрограммы, в головном модуле макрос `syscall` (а, значит, и включение файла `syscall.inc`) нам не понадобится. Текст выдаваемых программой сообщений мы опишем, как обычно, в виде инициализированных строк в секции `.data`; надо только не забывать, что в этой программе все строки должны иметь ограничивающий их нулевой байт. Буфер для чтения строки мы разместим в секции `.bss`. Что касается секции `.text`, то она будет состоять из сплошных вызовов подпрограмм.

```
global  _start              ; это головной модуль
extern  putstr              ; он использует подпрограммы
extern  getstr              ;   putstr, getstr и quit
extern  quit

section .data               ; описываем текст сообщений
nmq     db      'Hi, what is your name?', 10, 0
pmy     db      'Pleased to meet you, dear ', 0
exc     db      '!', 10, 0

section .bss                ; выделяем память под буфер
```

```
buf     resb    512
buflen  equ     $-buf

section .text
_start:                     ; начало головной программы
        push dword nmq      ; вызываем putstr для nmq
        call putstr
        add esp, 4
        push dword buflen   ; вызываем getstr
        push dword buf      ;   с параметрами buf и
        call getstr         ;   buflen
        add esp, 8
        push dword pmy      ; вызываем putstr для pmy
        call putstr
        add esp, 4
        push dword buf      ; вызываем putstr для
        call putstr         ;   строки, введённой
        add esp, 4          ;   пользователем
        push dword exc      ; вызываем putstr для exc
        call putstr
        add esp, 4
        call quit           ; вызываем quit
```

Итак, в нашей рабочей директории теперь находятся файлы `syscall.inc`, `strlen.asm`, `putstr.asm`, `getstr.asm`, `quit.asm` и `greet.asm`. Чтобы получить рабочую программу, нам понадобится отдельно вызвать NASM для каждого из модулей (напомним, что `syscall.inc` модулем не является):

```
nasm -f elf -dOS_LINUX strlen.asm
nasm -f elf -dOS_LINUX putstr.asm
nasm -f elf -dOS_LINUX getstr.asm
nasm -f elf -dOS_LINUX quit.asm
nasm -f elf -dOS_LINUX greet.asm
```

Отметим, что флажок `-dOS_LINUX` необходим только для тех модулей, которые используют `syscall.inc`, так что мы могли бы при компиляции `strlen.asm` и `greet.asm` его не указывать. Однако практика показывает, что проще указывать такие флажки всегда, нежели чем помнить, для каких модулей они нужны, а для каких — нет.

Результатом работы NASM станут пять файлов с суффиксом `.o`, представляющие собой *объектные модули* нашей программы. Чтобы объ-

единить их в исполняемый файл, нам потребуется вызвать редактор связей `ld`:

```
ld greet.o strlen.o getstr.o putstr.o quit.o -o greet
```

Результатом на сей раз станет исполняемый файл `greet`, который мы, как обычно, запустим на исполнение командой

```
./greet
```

## § 5.4. Объектный код и машинный код

Из приведённых выше примеров видно, что каждый объектный модуль, кроме всего прочего, характеризуется списком символов (в терминах ассемблера — меток), которые он предоставляет другим модулям, а также списком символов, которые ему самому должны быть предоставлены другими модулями. Буквально переведя с английского языка названия соответствующих директив (`global` и `extern`), мы можем назвать такие символы «глобальными» и «внешними»; чаще, однако, их называют «экспортируемыми» и «импортируемыми».

Ясно, что при трансляции исходного текста ассемблер, видя обращение к внешней метке, не может заменить эту метку конкретным адресом, поскольку этот адрес ему не известен — ведь метка определена в другом модуле, которого ассемблер не видит. Таким образом, всё, что может сделать ассемблер — это оставить под такой адрес свободное место в итоговом коде и записать в объектный файл информацию, которая позволит редактору связей расставить все такие «пропущенные» адреса, когда их значения уже будут известны.

При ближайшем рассмотрении оказывается, что заменить метки конкретными адресами ассемблер не может не только в случае обращений к внешним меткам, но **вообще никогда**. Дело в том, что, коль скоро программа состоит из нескольких (сколько угодно) модулей, ассемблер при трансляции одного из них никак не может предугадать, каким по счёту этот модуль будет стоять в итоговой программе, какого размера будут все предшествующие модули и, таким образом, не может знать, в какой области памяти (даже виртуальной) будет располагаться тот код, который ассемблер в настоящее время генерирует.

С другой стороны, известно, что редактор связей не видит исходных текстов модулей, да и не может их видеть, поскольку предназначен для связи модулей, полученных различными компиляторами из исходных

текстов на, вполне возможно, разных языках программирования. Следовательно, вся информация, необходимая для окончательного превращения объектного кода в исполняемый машинный, должна быть записана в объектный файл.

Таким образом, объектный код, который получается в качестве результата ассемблирования, представляет собой некий «полуфабрикат» машинного кода, в котором вместо абсолютных (числовых) адресов находится некая информация о том, как эти адреса вычислить и в какие места кода их следует расставить.

Отметим, что информацию о символах, содержащихся в объектном файле, можно узнать с помощью программы `nm`. В качестве упражнения попробуйте применить эту программу к объектным файлам написанных вами модулей (либо модулей из приведённых выше примеров) и попытаться проинтерпретировать результаты.

## § 5.5. Библиотеки

Чаще всего программы пишутся не «с абсолютного нуля», как это в большинстве примеров делали мы, а используют комплекты уже готовых подпрограмм, оформленные в виде *библиотек*. Естественно, такие подпрограммы входят в состав модулей, а сами модули удобнее иметь в заранее откомпилированном виде, чтобы не тратить время на их компиляцию; разумеется, полезно иметь в доступности и исходные тексты этих модулей, но в заранее откомпилированной форме библиотеки используются чаще.

Вообще говоря, различают программные библиотеки разных видов; например, бывают библиотеки макросов, которые, естественно, не могут быть заранее откомпилированы и существуют только в виде исходных текстов. Здесь мы, однако, рассмотрим более узкое понятие, а именно то, что под термином «библиотека» понимается на уровне редактора связей.

С технической точки зрения библиотека подпрограмм — это файл, объединяющий в себе некоторое количество объектных модулей и, как правило, содержащий таблицы для ускоренного поиска имён символов в этих модулях.

Необходимо отметить одно важнейшее свойство объектных файлов: каждый из них может быть включён в итоговую программу **только целиком** либо не включён вообще. Это означает, например, что если вы объединили в одном модуле несколько подпрограмм, а кому-то по-

требовалась лишь одна из них, в исполняемый файл всё равно войдёт код всего вашего модуля (то есть всех подпрограмм). Это необходимо учитывать при разбиении библиотеки на модули; так, системные библиотеки, поставляемые вместе с операционными системами, компиляторами и т. п., обычно строятся по принципу «одна функция — один модуль».

Для построения библиотеки из отдельных объектных модулей необходимо использовать специально предназначенные для этого программы. В ОС Unix соответствующая программа называется `ar`. Изначально её предназначение не ограничивалось созданием библиотек (само название `ar` означает «архиватор»), так что при вызове программы необходимо указать с помощью параметра командной строки, чего мы от неё добиваемся. Так, если бы мы захотели объединить в библиотеку все модули программы `greet` (кроме, разумеется, главного модуля, который не может быть использован в других программах), это можно было бы сделать следующей командой:

```
ar crs libgreet.a strlen.o getstr.o putstr.o quit.o
```

Результатом станет файл `libgreet.a`; это и есть библиотека. После этого скомпоновать программу `greet` с помощью редактора связей можно, например, так:

```
ld greet.o libgreet.a
```

или так:

```
ld greet.o -l greet -L .
```

В отличие от монолитного объектного файла, библиотека, будучи упакованной в один файл, продолжает, тем не менее, быть именно *набором объектных модулей*, из которых редактор связей выбирает только те, которые ему нужны для удовлетворения неразрешённых ссылок. Подробнее об этом мы расскажем в следующем параграфе.

## § 5.6. Алгоритм работы редактора связей

Редактору связей в командной строке указывается список объектов, каждый из которых может быть либо объектным файлом, либо библиотекой, при этом объектные файлы могут быть заданы только по имени файла, тогда как библиотеки могут задаваться двумя способами: либо

явным указанием имени файла, либо — с помощью флага `-l` — указанием ***имени библиотеки***, которое может упрощённо пониматься как имя файла библиотеки, от которого отброшены префикс `lib` и суффикс `.a`[1]. Так, в примере из предыдущего параграфа файл библиотеки назывался `libgreet.a`, а соответствующее *имя библиотеки* представляло собой слово `greet`. При использовании флага `-l` редактор связей пытается найти файл библиотеки с соответствующим именем в системных директориях (`/lib`, `/usr/lib` и т. п.), но можно указать ему дополнительные директории с помощью флага `-L`; так, «`-L .`» означает, что следует сначала попробовать найти библиотеку в текущей директории, и лишь затем начинать поиск в системных директориях.

В своей работе редактор связей использует два *списка символов*: список известных (разрешённых, от английского *resolved*) символов и список *неразрешённых ссылок (unresolved links)*. В первый список заносятся символы, *экспортируемые* объектными модулями (в своих текстах на языке ассемблера NASM мы помечали такие символы директивой `global`), во второй список заносятся символы, к которым уже есть обращения (то есть имеются модули, *импортирующие* эти символы), но которые пока не встретились ни в одном из модулей в качестве экспортируемых.

Редактор связей начинает работу, инициализировав оба списка символов как пустые, и шаг за шагом продвигается слева направо по списку объектов, указанных в его командной строке. В случае, если очередным указанным объектом будет объектный файл, редактор связей «принимает» его в формируемый исполняемый файл. При этом все символы, экспортируемые этим модулем, заносятся в список известных символов; если некоторые из них присутствовали в списке неразрешённых ссылок, они оттуда удаляются. Объектный код из модуля принимается редактором связей к последующему преобразованию в исполняемый код и вставке в исполняемый файл.

Если же очередным объектом из списка, указанного в командной строке, окажется библиотека, действия редактора связей будут более сложными и гибкими, поскольку возможно, что принимать *все* составляющие библиотеку модули ни к чему. Прежде всего редактор связей сверится со списком неразрешённых ссылок; если этот список пуст, библиотека будет полностью проигнорирована как ненужная. Однако обычно список в такой ситуации не пуст (иначе программист не стал

[1] Мы здесь не рассматриваем случай так называемых разделяемых библиотек, файлы которых имеют суффикс `.so`; концепция динамической загрузки требует дополнительного обсуждения, которое выходит за рамки данного пособия.

бы указывать библиотеку), и следующим действием редактора связей становятся поочерёдные попытки найти в библиотеке такие модули, которые экспортируют один или несколько символов с именами, фигурирующими в текущем списке неразрешённых ссылок; если такой модуль найден, редактор связей «принимает» его, соответствующим образом модифицирует списки символов и начинает рассмотрение библиотеки снова, и так до тех пор, когда ни один из оставшихся в библиотеке непринятых модулей не будет пригоден для разрешения ссылок. Тогда редактор связей прекращает рассмотрение библиотеки и переходит к следующему объекту из списка.

Таким образом, из библиотеки берутся только те модули, которые нужны, чтобы удовлетворить потребности предшествующих модулей в импорте символов, плюс, возможно, такие модули, в которых нуждаются уже принятые модули из той же библиотеки. Так, при сборке программы `greet` из предыдущего параграфа редактор связей сначала принял из библиотеки `libgreet.a` модули `getstr`, `putstr` и `quit`, поскольку в них присутствовали символы, импортируемые ранее принятым модулем `greet.o`; затем редактор связей принял и модуль `strlen`, поскольку в нём нуждался модуль `putstr`.

Редактор связей выдаёт сообщения об ошибках и отказывается продолжать сборку исполняемого файла в двух основных случаях. Первый из них — это появление в очередном принимаемом модуле экспортируемого символа, который к этому моменту уже значится в списке известных; иначе говоря, два или более принятых к рассмотрению модуля экспортируют один и тот же символ[2]. Это называется ***конфликтом имён***. Второй случай ошибочной ситуации возникает, когда список объектов (модулей и библиотек) исчерпан, а список неразрешённых ссылок не опустел, то есть как минимум один из принятых модулей ссылается в качестве внешнего на символ, который так ни в одном из модулей и не встретился; такая ошибочная ситуация называется ***неопределённой ссылкой*** (англ. *undefined reference*).

Интересно, что **редактор связей никогда не возвращается назад** в своём движении по списку объектов, так что если некоторый модуль из состава библиотеки не был принят на момент, когда редактор до этой библиотеки добрался, то потом он не будет принят тем более, даже если в каком-либо из последующих модулей появится импортируе-

[2]Некоторые современные редакторы связей в угоду нерадивым программистам позволяют в некоторых случаях не считать конфликт имён ошибкой; это используется, например, компиляторами языка Си++. Постарайтесь, насколько возможно, не полагаться на подобные возможности.

мый символ, который можно было бы разрешить, приняв ещё модули из ранее обработанной библиотеки. Из этого факта имеется важное следствие: объектные модули следует указывать **раньше**, чем библиотеки, в которых эти модули нуждаются. Вторым важным следствием является то, что **библиотеки никогда не должны «перекрёстно» зависеть друг от друга**, то есть если одна библиотека использует возможности второй, то вторая не должна использовать возможности первой. Если подобного рода перекрёстные зависимости возникли, такие две библиотеки следует объединить в одну.

Наконец, можно сделать ещё один вывод. До тех пор, пока библиотеки вообще не зависят друг от друга, мы можем не слишком волноваться о порядке параметров для редактора связей: достаточно сначала указать в произвольном порядке все объектные файлы, составляющие нашу программу, а затем, опять-таки в произвольном порядке, перечислить все нужные библиотеки. Если же зависимости между библиотеками появляются, порядок их указания становится важен, и при его несоблюдении программа не соберётся. Таким образом, зависимости библиотек друг от друга, даже не перекрёстные, порождают определённые проблемы. Поэтому, прежде чем полагаться при разработке одной библиотеки на возможности другой, следует многократно и тщательно всё обдумать.

Знание принципов работы редактора связей пригодится вам не только (и не столько) в учебном программировании на языке ассемблера, но и в практической работе на языках программирования высокого уровня, в особенности на языках Си и Си++. Не принимая во внимание содержание этого параграфа, вы рискуете, с одной стороны, перегрузить свои исполняемые файлы ненужным (неиспользуемым) содержимым, а с другой — спроектировать свои библиотеки так, что даже сами начнёте в них путаться.

# Приложение А

# Текст файла stud_io.inc

## Версия для ОС Linux

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; system dependend part

; generic 3-param syscall
%macro _syscall_3 4
        push edx
        push ecx
        push ebx
        push %1
        push %2
        push %3
        push %4
        pop edx
        pop ecx
        pop ebx
        pop eax
        int 0x80
        pop ebx
        pop ecx
        pop edx
%endmacro

; syscall_exit is the only syscall we use that has 1 parameter
%macro _syscall_exit 1
        mov ebx, %1     ; exit code
```

```
        mov eax, 1      ; 1 = sys_exit
        int 0x80
%endmacro

;; system dependent part ends here
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: read bytes
%macro _syscall_read 3
        _syscall_3 3,%1,%2,%3
%endmacro

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: written bytes
%macro _syscall_write 3
        _syscall_3 4,%1,%2,%3
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro  PRINT 1
        pusha
        pushf
        jmp %%astr
%%str   db      %1, 0
%%strln equ     $-%%str
%%astr: _syscall_write 1, %%str, %%strln
        popf
        popa
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro  PUTCHAR 1
        pusha
        pushf
%ifstr %1
        mov     al, %1
%elifnum %1
        mov     al, %1
%elifidni %1,al
        nop
%elifidni %1,ah
        mov     al, ah
%elifidni %1,bl
```

```
        mov     al, bl
%elifidni %1,bh
        mov     al, bh
%elifidni %1,cl
        mov     al, cl
%elifidni %1,ch
        mov     al, ch
%elifidni %1,dl
        mov     al, dl
%elifidni %1,dh
        mov     al, dh
%else
        mov     al, %1  ; memory location such as [var]
%endif
        sub     esp, 2  ; reserve memory for buffer
        mov     edi, esp
        mov     [edi], al
        _syscall_write 1, edi, 1
        add     esp, 2
        popf
        popa
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro  GETCHAR 0
        pushf
        push    edi
        sub     esp, 2
        mov     edi, esp
        _syscall_read 0, edi, 1
        cmp     eax, 1
        jne     %%eof_reached
        xor     eax,eax
        mov     al, [edi]
        jmp     %%gcquit
%%eof_reached:
        xor     eax, eax
        not     eax             ; eax := -1
%%gcquit:
        add     esp, 2
        pop     edi
        popf
%endmacro
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro FINISH 0-1 0
        _syscall_exit %1
%endmacro
```

## Версия для FreeBSD

Эта версия отличается от предыдущей только определением макросов `_syscall3` и `_syscall_exit`, поэтому целиком мы её не приводим. Чтобы получить рабочий файл для ОС FreeBSD, возьмите вышеприведённый текст для ОС Linux и замените определения этих макросов на следующие:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; freebsd-specific things

; generic 3-param syscall
%macro _syscall_3 4
        push    %4
        push    %3
        push    %2
        mov     eax, %1
        push    eax
        int     0x80
        add     esp, 16
%endmacro

%macro _syscall_exit 1
        push    %1      ; exit code
        mov     eax, 1  ; 1 = sys_exit
        push    eax
        int     0x80
        ; no cleanup - this will never return anyway
%endmacro

;; system dependent part ends here
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

# Литература


[1] Баурн С. Операционная система UNIX. М.:Мир, 1986.

[2] Робачевский А. М. Операционная система UNIX. Изд-во «BHV», Санкт-Петербург, 1997.

[3] Э. Танненбаум. Архитектура компьютера. 4-е издание. СПб.: Питер, 2003.

[4] The Netwide Assembler: NASM. `http://www.nasm.us/doc/` *Имеется русский перевод, выполненный AsmOs group; см. например,* `http://opslab.org.ru/nasm`

# Содержание

На официальном сайте А. В. Столярова
http://www.stolyarov.info
можно найти электронные версии этой и других книг автора.

Домашняя страница этой книги, расположенная по адресу http://www.stolyarov.info/books/asm_unix, содержит дополнительные материалы, в том числе тексты программ, упоминаемых в пособии.