Версия 0.91 pre-release от 07.11.2016. Возможны незначительные изменения.

Тема 13. Архитектуры вычислительных машин и языки ассемблера, часть 2

13.1. Циклы....1
13.2. Логическая (побитовая) арифметика....2
13.3 Основные конвенции вызовов подпрограмм....3
13.4 Прочие полезности....5
13.5 «Плавающая» арифметика....7
13.6 Системные вызовы....8
13.7 Работа с файлами....9
13.8 Работа со структурами данных....13
Литература и дополнительные источники к Теме 13....16

Обсуждаются вопросы, перечисленные на СЛАЙДЕ 1.

13.1. Циклы

Команда *loop*

Синтаксис (СЛАЙД 2):

```
loop  метка
```

Принцип работы:

- уменьшить значение регистра %*ecx* на 1;
- если %*ecx* = 0, то передать управление идущей после *loop* команде;
- если %*ecx* <> 0, то передать управление на метку.

Напишем программу для вычисления суммы чисел от 1 до 100 (конечно же, воспользовавшись формулой суммы арифметической прогрессии).

Код программы приведен на СЛАЙДЕ 3.

Организация произвольных циклов

На СЛАЙДАХ 4-5 приведен код программы для поиска наибольшего элемента массива.

Сначала в регистр %*eax* заносится число 0. После этого мы сравниваем каждый элемент массива с текущим наибольшим значением из %*eax*, и, если этот элемент больше, он становится текущим наибольшим. После просмотра всего массива в %*eax* находится наибольший элемент.

Код очень близкой по смыслу программы на C приведен на СЛАЙДЕ 6.

Не всегда такой способ обхода массива уместен. В C мы можем использовать переменную с номером текущего элемента, а не указатель на него. Никто не запрещает пойти этим же путём и на ассемблере, как в коде на СЛАЙДАХ 7-8.

В этом коде мы показали, как создавать произвольные циклы с постусловием на ассемблере, наподобие *do-while* в C. Ещё раз повторим эту конструкцию, выкинув весь код, не относящийся к циклу (СЛАЙД 9).

В C есть ещё цикл с предусловием *while*. Слегка изменив предыдущий код, получаем следующее (СЛАЙД 10).

Для особо дотошных слушателей, отметим (СЛАЙД 11), что цикл

```
for(init; cond; incr)
{
  body;
}
```

эквивалентен такой конструкции:

```
init;
while(cond)
{
  body;
  incr;
}
```

Таким образом, нам достаточно и уже рассмотренных двух видов циклов.

13.2. Логическая (побитовая) арифметика.

Кроме выполнения обычных арифметических вычислений, можно проводить и логические, то есть битовые (СЛАЙД 12). Они используются и для реализации операторов выбора из двух альтернатив (*if* с необязательным *else*), и для операторов циклов.

```
and   источник, приёмник
or    источник, приёмник
xor   источник, приёмник
not   операнд
test  операнд_1, операнд_2
```

Команды *and*, *or* и *xor* ведут себя так же, как и такие операторы языка С, как &, |, ^. Эти команды устанавливают флаги согласно результату.

Команда *not* инвертирует каждый бит операнда (изменяет на противоположный), так же как и оператор языка С ~.

Команда *test* выполняет побитовое И над операндами, как и команда *and*, но, в отличие от неё, операнды не изменяет, а только устанавливает флаги. Её также называют командой логического сравнения, потому что с её помощью удобно проверять, установлены ли определённые биты. Например, так, как показано на СЛАЙДЕ 13 (верхняя часть).

Для записи константы в двоичной системе счисления: используется префикс 0*b*.

Команду *test* можно применять для сравнения значения регистра с нулём (нижняя часть СЛАЙДА 13).

К логическим командам также можно отнести команды сдвигов.

Сдвиги

К логическим командам также можно отнести команды сдвигов (СЛАЙД 14):

```
/* Shift Arithmetic Left/SHift logical Left */
sal/shl количество_сдвигов, назначение

/* SHift logical Right */
shr     количество_сдвигов, назначение

/* Shift Arithmetic Right */
sar     количество_сдвигов, назначение
```

Количество_сдвигов может быть задано непосредственным значением или находиться в регистре %*cl*. Учитываются только младшие 5 бит регистра %*cl*, так что количество сдвигов может варьироваться в пределах от 0 до 31.

Принцип работы команды *shl* показан на СЛАЙДЕ 15.

Принцип работы команды *shr* показан на СЛАЙДЕ 16.

Эти две команды называются **командами логического сдвига**, потому что они работают с операндом как с массивом битов. Каждый «выдвигаемый» бит попадает в флаг *cf*, причём с другой стороны операнда «вдвигается» бит 0. Таким образом, в флаге *cf* оказывается самый последний «выдвинутый» бит. Такое поведение вполне допустимо для

работы с беззнаковыми числами, но числа со знаком будут обработаны неверно из-за того, что знаковый бит может быть потерян.

Для работы с числами со знаком существуют команды арифметического сдвига. Команды *shl* и *sal* выполняют полностью идентичные действия, так как при сдвиге влево знаковый бит не теряется (расширение знакового бита влево становится новым знаковым битом). Для сдвига вправо применяется команда *sar*. Она «вдвигает» слева знаковый бит исходного значения, таким образом, сохраняя знак числа. Принцип их работы показан на СЛАЙДЕ 17.

Многие программисты на языке С знают об умножении и делении на степени двойки (2, 4, 8…) при помощи сдвигов. Этот фокус отлично работает и в ассемблере, его можно использовать для оптимизации.

Кроме сдвигов обычных, существуют циклические сдвиги (СЛАЙД 18):

```
/* ROtate Right */
ror   количество_сдвигов, назначение

/* ROtate Left */
rol   количество_сдвигов, назначение
```

При циклическом сдвиге влево на три бита: три старших («левых») бита «выдвигаются» из регистра влево и «вдвигаются» в него справа. При этом в флаг *cf* записывается самый последний «выдвинутый» бит.

Принцип работы команды *rol* показан на СЛАЙДЕ 19.

Принцип работы команды *ror* показан на СЛАЙДЕ 20.

Существует ещё один вид сдвигов – циклический сдвиг через флаг *cf*. Эти команды рассматривают флаг *cf* как продолжение операнда (СЛАЙД 21).

```
/* Rotate through Carry Right */
rcr   количество_сдвигов, назначение

/* Rotate through Carry Left */
rcl   количество_сдвигов, назначение
```

Принцип работы команды *rcl* показан на СЛАЙДЕ 22, а команды *rcr* — на СЛАЙДЕ 23. Эти сложные циклические сдвиги редко используются на практике, но уже сейчас нужно знать, что такие инструкции существуют, чтобы не изобретать колесо снова. Ведь в языке С циклический сдвиг производится примерно так, как показано на СЛАЙДЕ 24.

13.3 Основные конвенции вызовов подпрограмм

Несмотря на подробное описание механизма стековых фреймов, сделанное ранее, в некоторых аспектах остается возможность для маневра. Так, например, в каком порядке следует заносить в стек значения фактических параметров? Когда мы пишем программу на языке ассемблера, этот вопрос, собственно говоря, не возникает. Однако он может оказаться неожиданно принципиальным при создании компиляторов языков программирования высокого уровня.

Создатели компиляторов языка Паскаль и ему подобных языков идут, казалось бы, очевидным путем: вызов процедуры или функции транслируется в серии команд занесения в стек значений, которые вталкиваются туда в естественном порядке – слева направо. Затем в код вставляется команда *call*. Когда такая подпрограмма получает управление, значения фактических параметров располагаются в стеке снизу вверх, т.е. последний переданный параметр оказывается размещенным ближе других к основанию стекового фрейма (доступен по адресу 8(%*ebp*)). Следовательно, для доступа к первому и всем последующим параметрам процедуры или функции языка Паскаль необходимо знать общее количество этих параметров, т.к. расположение *i*-го параметра в стековом фрейме зависит от общего количества. Так, если у процедуры три 4-байтных параметра, то первый

из них окажется в стеке по адресу 16(%*ebp*), если их шесть, то первый придется искать по адресу 28(%*ebp*). Именно поэтому язык Паскаль не допускает процедуры или функции с переменным числом аргументов, иногда называемых еще **вариадическими подпрограммами**.

Это вполне нормально для языка обучения программированию, но не всегда приемлемо для профессионального языка. Создатели Паскаля поступили хитроумно, разрешив псевдопроцедуры с переменным числом аргументов, например, *ReadLn*. На самом деле они являются частью самого языка, компилятор трансформирует их вызовы в некое представление, весьма далекое от вызова подпрограммы на уровне машинного кода. Для программиста вывод очевиден: на Паскале нельзя описать процедуру подобного рода.

Создатели языка С пошли по другому пути. При трансляции вызова функции языка С параметры помещаются в стек в обратном порядке и оказываются размещенными во фрейме в порядке сверху вниз. Первый параметр всегда оказывается доступным по адресу 8(%*ebp*), второй – 12(%*ebp*) и т.д., это не зависит от количества аргументов. Понятно, что если функция вызвана без параметров, то никакого первого и последующих параметров в стеке не будет. Это позволяет создавать функции с переменным числом аргументов. Кроме того, в язык С не входит ни одной функции, все они являются частью библиотеки, хотя и написаны, как правило, на С.

Отсутствие в Паскале подпрограмм с переменным числом аргументов позволяет возложить заботы об очистке стека на вызываемую сторону. Программа на Паскале всегда знает, сколько места занимают фактические параметры в стековом фрейме, т.к. количество параметров задано и неизменно. Значит, она может принять на себя заботу о стеке. Поскольку вызовов в программе больше, чем самих подпрограмм, то за счет перекладывания заботы об очистке стека с вызывающей стороны на вызываемую достигается определенная экономия памяти (по количеству машинных команд).

При использовании соглашений языка С такая экономия невозможна, поскольку в общем случае подпрограмма не знает о количестве параметров. Ответственность за очистку стека от параметров остается на вызывающей стороне. Обычно это делается простым увеличением значения %*esp* на число, равное совокупной длине фактических параметров. Например, если подпрограмма *proc* принимает на вход три 4-байтных параметра *p*1, *p*2 и *p*3, то ее вызов будет выглядеть примерно так, как показано на СЛАЙДЕ 25.

В случае использования соглашения языка Паскаль последняя команда *add* оказывается ненужной. Обо всем позаботится вызываемая сторона. В системе команд x86 есть на этот случай специальная форма команды *ret* с одним операндом. Этот операнд, который может быть только непосредственным и всегда имеет длину два байта (*word*), задает количество памяти в байтах, занятой параметрами подпрограммы. Например, процедуру, принимающую три 4-байтных параметра, компилятор языка Паскаль закончит командой *ret 12*.

Эта команда, как и обычная команда *ret*, извлечет из стека адрес возврата и передаст по нему управление, и одновременно с этим увеличит значение %*esp* на заданное число, избавляя вызвавшую сторону от обязанности по очистке стека.

Рекурсии

Как известно, рекурсивный вызов, это когда подпрограмма прямо или косвенно вызывает сама себя. Глубина вызовов, как правило, является системной константой. Начинающим программистам обычно демонстрируют рекурсию на примере вычисления факториала по известной формуле: 0! = 1; $N! = N*(N-1)!$ для $N > 0$. Код приведен на СЛАЙДЕ 26 и 27.

Однако некоторыми специалистами (но не математиками) такая демонстрация рекурсии справедливо критикуется, поскольку существует более элегантное решение, где используются циклы. В нем просто перемножаются все числа от 1 до *N*. Код программы

на ассемблере приводится на СЛАЙДЕ 28.

Видно, что рекурсия переписана в виде цикла. Стековый фрейм больше не нужен, т.к. стек не нужен (в него ничего не помещается), и другие функции не вызывается. Пролог и эпилог по этой причине удалены, при этом регистр %*ebp* не используется вообще. Но если бы он использовался, то сначала, конечно же, нужно было бы сохранить его значение, а перед возвратом восстановить.

13.4 Прочие полезности

Обмен значениями

Команда *xchg* производит обмен значениями двух операндов и имеет очень простой синтаксис (СЛАЙД 29):

```
xchg     операнд_1, операнд_2
```

При этом один из операндов или оба должны быть регистрами. В первом случае для доступа к другому операнду используется непосредственная адресация. Пример кода взаимного обмена двух нерегистровых значений приведен на СЛАЙДАХ 30 и 31.

Команда *bswap* позволяет изменять порядок байтов в 32-битном длинном слове. СЛАЙД 32.

Операции с цепочками

При обработке данных часто приходится иметь дело с цепочками данных. Цепочка, как подсказывает название, представляет собой массив данных – несколько переменных одного размера, расположенных друг за другом в памяти. В языке С мы используем массив и индексирование, например, *argv*[*i*]. Но в ассемблере для последовательной обработки цепочек есть специализированные команды. Синтаксис (СЛАЙД 33):

```
lods

stos
```

Откуда же эти команды знают, где брать данные и куда их записывать? Ведь у них нет аргументов! Тут самое время вспомнить про регистры %*esi* и %*edi* и про их немного странные названия: «*индекс источника*» (*source index*) и «*индекс приёмника*» (*destination index*). Так вот, все цепочечные команды подразумевают, что в регистре %*esi* находится указатель на следующий необработанный элемент цепочки-источника, а в регистре %*edi* – указатель на следующий элемент цепочки-приёмника .

Направление просмотра цепочки задаётся флагом *df*: 0 – просмотр вперед, 1 – просмотр назад.

Команда *lods* загружает элемент из цепочки-источника в регистр %*eax*/%*ax*/%*al* (размер регистра выбирается в зависимости от суффикса команды). После этого значение регистра %*esi* увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Команда *stos* записывает содержимое регистра %*eax*/%*ax*/%*al* в цепочку-приёмник. После этого значение регистра %*edi* увеличивается или уменьшается (в зависимости от направления просмотра) на значение, равное размеру элемента цепочки.

Пример программы, работающей с цепочечными командами, приведен на СЛАЙДАХ 34 и 35. Она увеличивает каждый байт строки *str_in* на 1, то есть заменяет *a* на *b*, *b* на *c* и т.д.

Но над цепочками мы часто выполняем стандартные действия. Например, при копировании блоков памяти мы просто пересылаем байты из одной цепочки в другую, без обработки. При сравнении строк мы сравниваем элементы двух цепочек. При вычислении

длины строки в языке С мы считаем байты до тех пор, пока не встретим нулевой байт. Эти действия очень просты, но, в тоже время, используются очень часто, поэтому были введены следующие команды (СЛАЙД 36):

```
movs

cmps

scas
```

Размер элементов цепочки, которые обрабатывают эти команды, зависит от использованного суффикса команды.

Команда *movs* выполняет копирование одного элемента из цепочки-источника в цепочку-приёмник.

Команда *cmps* выполняет сравнение элемента из цепочки-источника и цепочки-приёмника (фактически, как и *cmp*, выполняет вычитание, источник – приёмник, результат никуда не записывается, но флаги устанавливаются).

Команда *scas* предназначена для поиска определённого элемента в цепочке. Она сравнивает содержимое регистра %*eax*/%*ax*/%*al* и содержимое элемента цепочки (выполняется вычитание %*eax*/%*ax*/%*al* – элемент_цепочки, результат не записывается, но флаги устанавливаются). Адрес цепочки должен быть помещён в регистр %*edi*.

После того, как эти команды выполнили своё основное действие, они увеличивают/уменьшают индексные регистры на размер элемента цепочки.

Данные команды обрабатывают только один элемент цепочки. Таким образом, нужно организовать что-то вроде цикла для обработки всей цепочки. Для этих целей существуют **префиксы команд** (СЛАЙД 37):

```
rep

repe/repz

repne/repnz
```

Эти префиксы ставятся перед командой, например: *repe scas*. Префикс организовывает как бы цикл из одной команды, при этом с каждым шагом цикла значение регистра %*ecx* автоматически уменьшается на 1.

rep повторяет команду, пока %*ecx* не равен нулю.

repe (или *repz* – то же самое) повторяет команду, пока %*ecx* не равен нулю и установлен флаг *zf*. Анализируя значение регистра %*ecx*, можно установить точную причину выхода из цикла: если %*ecx* равен нулю, значит, *zf* всегда был установлен, и вся цепочка пройдена до конца, если %*ecx* больше нуля — значит, флаг *zf* в какой-то момент был сброшен.

repne (или *repnz* – то же самое) повторяет команду, пока %*ecx* не равен нулю и не установлен флаг *zf*.

Также укажем команды (СЛАЙД 38) для управления флагом *df*:

```
cld

std
```

cld (*CLear Direction flag*) сбрасывает флаг *df*.

std (*SeT Direction flag*) устанавливает флаг *df*.

Пример кода для вновь создаваемой функции *my_memcpy* приведен на СЛАЙДАХ 39 и 40.

Еще один пример для функции *my_strlen* приведен на СЛАЙДАХ 41 и 42. Нам

нужно сравнить каждый байт цепочки с 0, остановиться, когда найдём 0, и вернуть количество ненулевых байтов. В качестве счетчика мы будем использовать регистр %*ecx*, который автоматически изменяют все префиксы. Но префиксы уменьшают счетчик и прекращают выполнение команды, когда %*ecx* равен 0. Поэтому перед цепочечной командой мы поместим в %*ecx* число 0*xffffffff*, и этот регистр будет уменьшаться в ходе выполнения цепочечной команды. Результат получится в обратном коде, поэтому мы используем команду *not* для инвертирования всех битов. И после этого ещё уменьшим результат на 1, так как нулевой байт тоже был посчитан.

В заключение рассмотрения цепочечных команд нужно сказать следующее: не следует заново изобретать стандартные функции, как мы только что сделали. Это всего лишь пример и объяснение принципов их работы. В реальных программах используйте цепочечные команды, только когда они реально смогут помочь при нестандартной обработке цепочек, а для стандартных операций лучше вызывать библиотечные функции.

13.5 «Плавающая» арифметика

Для выполнения действий над числами с плавающей точкой в x86_32 имеются несколько вариантов. Мы коротко рассмотрим один из них – микропроцессорный блок x87 (*Floating Point Unit*, далее – FPU). Блок FPU «инкапсулирует» дополнительные к основной системе инструкций команды, дополнительные регистры и исполнительные устройства. Регистры FPU показаны на СЛАЙДЕ 43. Они, а также использующие их инструкции позволяют выполнять довольно сложные математические функции, которые могут потребоваться для графических задач, цифровой обработке сигналов, бизнес-приложений.

В программах можно использовать три типа «плавающих» чисел. СЛАЙД 44. Числа в памяти хранятся в «интеловском» формате. Элементы массивов хранятся в том порядке, в котором записывались значения соответствующей директивой. Для чисел одинарной точности используется директива *.float*, для чисел двойной точности - *.double*, для 80-битных значений - *.tfloat*.

Семейство инструкций

```
fst/fld source
```

используется для перемещения плавающих значений в/из регистров FPU. В качестве *source* может выступать любое 32-, 64- и 80-битная область памяти. Код, представленный на СЛАЙДЕ 45, демонстрирует, как объявляются и используются в программе плавающие числа.

Для чисел одинарной точности используются инструкции *flds* и *fsts*, для двойной – *fsts* и *fstl*.

На СЛАЙДЕ 46 показаны некоторые предустановленные плавающие значения, которые могут загружаться в регистры FPU. С их помощью можно выполнять действия над некоторыми общеизвестными математическими константами. Некоторую странность может представлять инструкция *fldz*. В «плавающих» типов данных следует различать +0.0 и -0.0. В большинстве операций они одинаковы, но дают различные результаты, например, при делении (плюс бесконечность и минус бесконечность).

Регистровый стек FPU

В FPU, как указывалось выше, 11 регистров. Из них 8 – это 80-битные регистры данных, а оставшиеся – это регистр управления, регистр состояния и так называемый тэговый регистр.

Регистры данных называются *R*0 – *R*7, хотя из программ к ним обращаются по другим именам. См. СЛАЙД 47. Следует отметить, что это особенный стек, поскольку он циклический: нижний его элемент указывает на верхний. Вершиной стека обычно

является регистр *st*0, но проверить это можно, узнав содержимое слова состояния FPU. Все остальные регистры определяются смещением по отношению к вершине: *st*1 – *st*7.

Базовая арифметика FPU

Как и ожидалось, FPU позволяет выполнять базовые математические операции. В общем виде они представлены на СЛАЙДЕ 48. Очевидно, что каждая из них должна предоставлять, как минимум, шесть вариантов. Например, для сложения это выглядит так, как показано на СЛАЙДЕ 48. Некоторые примеры базовой арифметики приведены на СЛАЙДЕ 49.

Рассмотрим более полный пример для выражения ((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) – (140.2 / 94.21)). Это займет 13 шагов.

1. Загрузить 43.65 в ST0.
2. Делить ST0 на 22, результат в ST0.
3. Загрузить 76.34 в ST0 (результат шага 2 перемещается в ST1).
4. Загрузить 3.1 в ST0 (результата шага 3 перемещается в ST1, а шага 2 — в ST2).
5. Умножить ST0 на ST1, результат в ST0.
6. Прибавить к ST0 содержимое ST2, результат в ST0 (это левая часть нашего «равенства»).
7. Загрузить 12.43 в ST0 (результат шага 6 перемещается в ST1).
8. Умножить ST0 на 6,оставляя результат в ST0.
9. Загрузить 140.2 в ST0 (результат шага 8 перемещается в ST1, а шага 6 — в ST2).
10. Загрузить 94.21 в ST0 (результат шага 8 перемещается в ST2, а шага 6 — в ST3).
11. Разделить ST1 на содержимое ST0, выталкивая элемент из стек и сохраняя результат в ST0 (результат шага 8 перемещается в ST1, а шага 6 — в ST2).
12. Вычесть ST0 из ST1, результат в ST0 (это правая часть нашего «равенство»).
13. Разделить ST2 на ST0, сохраняя результат в ST0 (это наше решение).

Содержимое стека FPU на каждом из шагов показано на СЛАЙДЕ 50. Код программы на СЛАЙДАХ 51-52.

Дополнительная арифметика FPU

При решении многих задач имеется возможность использовать не только базовые арифметические операции, но и некоторые другие функции. Они показаны на СЛАЙДЕ 53. Названия большинства из этих самоописываемые.

Операции сравнения

К сожалению, сравнение плавающих значений не так просто, как в целочисленных операциях. В последних достаточно просто посмотреть содержимое флагового регистра и в зависимости от результата передавать управление той или иной инструкции. Плавающие сравнения используют регистр статуса. Некоторые из инструкций, а также содержимое битов кодов условий в регистре состояния FPU показаны на СЛАЙДЕ 54.

13.6 Системные вызовы

Программа, которая не взаимодействует с внешним миром, вряд ли может сделать что-то полезное. Вывести сообщение на экран, прочитать данные из файла, установить сетевое соединение – это примеры действий, которые программа не может совершить без помощи ОС. В Linux прикладной программный интерфейс ядра организован через системные вызовы. Системный вызов можно рассматривать как функцию, которую для клиента выполняет ОС.

Теперь наша задача состоит в том, чтобы разобраться, как происходит системный вызов. За каждым системным вызовом закреплен свой номер. Все они (для 32-битных систем) перечислены в файле */usr/include/asm/unistd_32.h*. См. СЛАЙД 55.

Системные вызовы считывают свои параметры из регистров. Номер системного вызова нужно поместить в регистр %*eax*. Параметры помещаются в остальные регистры в таком порядке:

первый – в %*ebx*;
второй – в %*ecx*;
третий – в %*edx*;
четвертый – в %*esi*;
пятый – в %*edi*;
шестой – в %*ebp*.

Следовательно, используя все доступные регистры общего назначения, можно передать не более 6 параметров. В Linux системный вызов производится вызовом прерывания с номером 0x80. Такой способ вызова с передачей параметров через регистры называется быстрым (*fastcall*). В других системах (FreeBSD, *Solaris* / *OpenIndiana*) могут применяться другие способы вызова. В частности, в семействе *BSD параметры системного вызова передаются через стек согласно конвенции языка C, а об успешном или неуспешном выполнении приходится судить по содержимому бита *CF* регистра %*eflags*.

Очевидно, что не следует использовать системные вызовы везде, где только можно, без острой необходимости. В разных версиях ядра порядок аргументов у некоторых системных вызовов отличается. Это может приводить к ошибкам, которые не всегда легко найти. Поэтому стоит использовать функции стандартной библиотеки C, ведь их сигнатуры не изменяются, и это обеспечивает переносимость кода на C. Поэтому до сих пор мы пользовались этим, и с некоторой долей условности писали переносимые ассемблерные программы. Только если по каким-то причинам приходится писать небольшой участок высоконагруженного кода, и для него недопустимы накладные расходы, вносимые вызовом стандартной библиотеки C, – только тогда стоит использовать системные вызовы напрямую.

13.7 Работа с файлами

В каждой ОС есть свой собственный способ работы с файлами. Однако, тот способ, который используется в *NIX-системах, включая Linux, считается наиболее простым и универсальным. *NIX-файлам безразлично, какие программы их создают; доступ к ним осуществляется как к последовательному потоку байт. Когда нам нужен доступ к файлу, мы начинаем с открытия его по имени. ОС дает нам число, называемое **файловым дескриптором**, который мы используем в качестве ссылки на файл до тех пор, пока проходим через него. Мы читаем и пишем в файл через его дескриптор. По окончании чтения или записи, мы закрываем файл. Это приводит к тому, что файловый дескриптор становится бесполезным.

В наших программах мы работаем с файлами следующими способами (СЛАЙД 56):

1. Сообщаем системе имя открываемого файла и то, в каком режиме он должен быть открыт – читаем, пишем, читаем-пишем, создаем, если он не существует, и т.д. Это обрабатывается системным вызовом *open*, который принимает в качестве параметров число, представляющее собой режим доступа, и набор разрешенных и запрещенных операций над файлом. Регистр %*eax* хранит номер системного вызова, который равен 5. Адрес первого символа в имени файла должен быть записан в %*ebx*. Режим доступа в числовом формате – в %*ecx*. Для того чтобы читать из файлов, используется 0, а 03101 предназначен для файлов, в которые мы хотим писать (обязательно указывается лидирующий ноль). Наконец, набор разрешений, также записывается как число в регистр %*edx*. Для тех, кто не знаком с наборами разрешений в *NIX, можно просто указать 0666 (как водится с лидирующим нулем). По традиции здесь используется 8-ричная система счисления, но никто, разумеется, не запрещает 10- и 16-ные системы.

2. Номер файлового дескриптора будет возвращен системным вызовом через регистр %*eax*. А это, напомним, то число, которое используется для работы с файлами.

3. Далее можно оперировать с файлом, выполняя чтение и/или запись, всякий раз указывая ОС номер дескриптора. Чтение (*read*) – это системный вызов #3, и чтобы его вызвать, нужно разместить файловый дескриптор в регистре %*ebx*, адрес буфера для хранения данных для чтения – в регистре %*ecx*, а его размер – в %*edx*. Буферы объявляются обычно в секции с меткой *.bss*. Вызов *read* возвратит либо количество прочитанных из файла байт или код ошибки. Коды ошибок можно определить, потому что они всегда являются отрицательными числами. Запись (*write*) – это системный вызов #4, и его параметры почти идентичны вызову *read* за исключением того, что буферы должны быть заполнены данными для записи. Этот системный вызов возвратит результат через регистр %*eax*.

4. По окончании работы с файлами, мы должны сообщить ОС об их закрытии. После этого наш файловый дескриптор не является валидным. Это делается с помощью системного вызова *close* (#6). Очевидно, что единственный его параметр это номер файлового дескриптора, размещаемый в регистре %*ebx*.

Под буфером понимается непрерывный блок байтов, используемых для нетипизированного переноса данных. Когда мы запрашиваем чтение данных, ОС необходимо место для размещения этих данных. Это место и называется **буфером**. Обычно они используются для временного хранения данных, затем программа считывает данные из буферов и преобразует их в форму, удобную для дальнейшего использования. Однако наши программы не настолько сложны, чтобы делать это. Например, давайте считать, что нам нужно прочитать одну строку текста из файла, но мы не знаем ее размера. В этом случае мы должны просто читать большое количество байт/символов из файла в буфер, найти символ конца строки и скопировать все до этого символа в другое место. Если мы не найдем символ конца строки, нужно распределить дополнительную память для другого буфера и продолжить чтение. Вероятнее всего нам придется иметь дело с теми символами, которые остались в старом буфере и использовать их в качестве отправной точки при работе с новым буфером.

Кроме того, важно отметить, что буферы имеют фиксированный размер, устанавливаемый программистом. То есть, если нам нужно прочитать 500 байт за один раз, мы передаем системному вызову *read* адрес буфера из неиспользуемой области и передаем число 500, чтобы вызов знал, насколько велик этот буфер. Мы можем сделать его меньше или больше при необходимости. В коде программы это будет выглядеть так, как показано в СЛАЙДЕ 57.

Важно отметить, что при старте Linux-программы обычно открытыми являются по крайней мере три дескриптора (СЛАЙД 58). Это:

STDIN – стандартный ввод, файл только для чтения, и обычно является синонимом клавиатуры, файловый дескриптор равен 0.

STDOUT – стандартный вывод, файл только для записи, и обычно является синонимом экрана дисплея, файловый дескриптор равен 1.

STDERR – стандартный вывод ошибок. Так же файл только для записи, и обычно является синонимом экрана дисплея. Большая часть вывода идет на STDOUT, но любые сообщения об ошибках направляются в STDERR. Таким образом, при необходимости можно направить их, так сказать, в разные стороны. Файловый дескриптор равен 2.

Любой из этих файлов может быть перенаправлен из/в реальный файл, отличный от экрана и клавиатуры. Программам самим по себе нет нужды беспокоиться о таком перенаправлении, они могут использовать стандартные файловые дескрипторы обычным образом.

Теперь мы попробуем написать программу, чтобы проиллюстрировать вышеизложенное (СЛАЙД 59). Программа будет работать с двумя файлами, читать из первого, переводить все строчные буквы в прописные и записывать их в другой файл.

Сначала стоит подумать, как нам нужно организовать работу, чтобы она была выполненной:

- Функция, которая принимает блок памяти и преобразует его в «верхний регистр». Как параметры, ей нужны будут адрес блока памяти и его размер.
- Секция кода, которая циклически считывает данные из входного файла, размещая их в буфере, вызывает нашу функцию, и затем измененный буфер записывает в выходной файл.
- Начало функционирования программы с открытием необходимых файлов.

Мы можем посчитать, что никогда не запомним все эти номера – системных вызовов, прерываний и т.д. В нашей программе мы с помощью директивы *.equ* свяжем имена с номерами (СЛАЙД 60). Например, если мы напишем:

```
.equ LINUX_SYSCALL, 0x80 ,
```

то всякий раз при использовании *LINUX_SYSCALL*, ассемблер будет подставлять вместо него 0x80. Теперь мы можем, в частности, написать

```
int $LINUX_SYSCALL
```

Это несколько удобнее для чтения и запоминания.

Далее приведен код программы. Отметим, что в ней больше меток, чем мы в действительности используем для переходов, поскольку часть из них введена для ясности. Стоит попытаться прочитать программу и посмотреть, что происходит в разных ситуациях. После кода мы приведем его подробное пояснение. См. СЛАЙДЫ с 61 по 69.

Попробуем разобраться, как программа работает.

Первая секция помечена как *CONSTANTS*. Как известно, константа – это значение присваиваемое объекту при ассемблировании, компиляции и иногда выполнении программы, которое никогда не изменяется. Мы разместили все константы в одном месте программы. Это является неплохим стилем, т.к. позволяет легко их находить. А то, что мы записали константы в верхнем регистре, отличает их от имен переменных и при необходимости позволяет перейти к секции констант.

Как мы и хотели, константы объявлены директивой *.equ*. В нашем случае в качестве констант мы продекларировали имена для всех «стандартных» чисел: номеров системных вызовов, номеров прерываний и параметров открытия файлов.

Следующая секция кода помечена как *BUFFERS*. У нас только один буфер, который мы назвали *BUFFER_DATA*. Также мы определили константу *BUFFER_SIZE* с вполне читабельным и понятным именем. При возможной модификации нам понадобится изменить только определение вместо того, чтобы искать в коде все числа, равные 500 и изменяя их.

Теперь перенесемся в конец исходного текста программы, где у нас реализована функция *convert_to_upper*. И она действительно делает преобразование.

Секция кода также начинается с констант, которые используются функцией. Сюда мы их разместили только по этой причине. У нас есть три определения:

```
.equ LOWERCASE_A, 'a'
.equ LOWERCASE_Z, 'z'
.equ UPPER_CONVERSION, 'A' - 'a'
```

Первые два определяют границы диапазон строчных латинских литер. Поскольку литеры на самом деле представляются числами, то над ними можно выполнять арифметические операции. Это мы использовали при определении константы

UPPER_CONVERISON.

Далее у нас объявлено несколько констант, помеченных как *STACK STUFF*, после которых указана точка входа в функцию *convert_to_upper* со стандартным прологом. После него две строки, копирующие параметры функции в нужные регистры.

Потом мы обнуляем регистр %*edi*. Мы собираемся организовать цикл по всем байтам нашего буфера путем загрузки по ссылке %*eax* + %*edi*, инкрементируя %*edi* и повторяя эти операции до тех пор, пока %*edi* не станет равным длине буфера, сохраненного в регистре %*ebx*.

Две строки:

```
cmpl $0, %ebx
je end_convert_loop
```

всего лишь проверка того, что не передан буфер нулевой длины. Если это сделано, мы просто освобождаем ресурсы и выходим из функции. В спецификации функции, конечно, можно указать, что нельзя передавать буфер нулевого размера, но такая проверка, как у нас, никогда не бывает, конечно, лишней.

Теперь функция может начать цикл. Первым делом она копирует байт в регистр %*cl*. Код:

```
movb (%eax,%edi,1), %cl
```

Здесь мы использовали индексную косвенную адресацию. В данном случае мы стартуем с %*eax*, прибавляем к номеру ячейки значение %*edi*, каждая ячейка размером 1 байт. Вычисленное таким образом значение размещается в регистре %*cl*. После этого проверяется, что значение находится в указанном диапазоне. Для этого мы всего лишь сравниваем код символа с кодом строчной литеры *a*. Если результат отрицательный, то это не может быть строчная буква. Аналогичное сравнение осуществляется и с буквой *z*, и в этом случае символ не может быть строчной буквой. В обоих случаях, мы просто начинаем новую итерацию цикла. Если же символ в нужном диапазоне, то прибавляется константа *UPPER_CONVERISON*, и измененное значение сохраняется обратно в буфер.

В любом случае, мы переходим к просмотру следующего элемента буфера инкрементированием %*cl*. Затем мы проверяем, не достигнут ли конец буфера. Если нет, то мы переходим обратно в начало цикла на метку *convert_loop*. Иначе, продолжаем выполнение до конца функции. Поскольку мы модифицируем буфер напрямую, нет нужды что-либо возвращать вызывающей стороне, т.к. все изменения уже зафиксированы в буфере. Указание метки *end_convert_loop* не обязательно, но таким образом мы показываем, в каком месте программы мы находимся.

Теперь мы знаем, как работает преобразование буфера, и можно перейти к рассмотрению того, как данные считываются и записываются из/в файлы.

Перед чтением и записью файлы должны быть открыты. За это, как известно, отвечает системный вызов *open*. Он принимает следующие параметры:

• %*eax* хранит номер системного вызова, в данном случае – 5.

• %*ebx* хранит указатель на строку с именем открываемого файла. Строка обязательно завершается символом с кодом 0.

• %*ecx* хранит опции, используемые для открытия файла, т.е. то, как открывать файл. Они показывают такие вещи, как открытие на чтение; открытие на запись; открытие на чтение и запись; создание, если файл не существует; удаление файла, если он уже существует и т.д.

• %*edx* содержит разрешения, которые используются для открытия файла. Традиционно они задаются числа в 8-ричной системе, но это обычные для *NIX-систем разрешения.

После выполнения системного вызова дескриптор вновь открытого файла

сохраняется в регистр %*eax*.

Какие же файлы открывает наша программа? Все просто. В нашем примере имена задаются через командную строку. К счастью, ее параметры ОС сохраняет в легко доступных местах, и они содержат маркеры конца строки, заданные символами с кодом 0. Когда *Linux*-программа стартует, все аргументы командной строки сохраняются в стеке. Количество аргументов находится 8(%*esp*), имя программ – в 12(%*esp*), а аргументы сохраняются, начиная с *16*(%esp).

В языке С для этой цели мы используем массив *argv*, такой же способ использует наша программа.

Первым делом наша программа сохраняет текущую позицию указателя стека в %*ebp*, а затем резервирует в нем пространство для файловых дескрипторов. После этого открываются файлы. Первый файл программа открывает как входной, его имя задается первым аргументом командной строки. Мы делаем это, чтобы создать параметры системного вызова. Мы размещаем имя файла в %*ebx*, режим «только для чтения» в %*ecx*, разрешения операций над файлами ($0666) в %*edx*, а номер системного вызова – в %*eax*. В этом регистре после выполнения системного вызова окажется номер файлового дескриптора. Затем он копируется в подходящее место в стеке.

Аналогичные действия выполняются для формирования выходного файла. Отличия: создается в режиме «только на запись», «создать, если не существует», «стереть содержимое, если файл существует». Этот файловый дескриптор также запоминается в стеке.

Теперь мы подошли к главному циклу чтения/записи. Коротко говоря, мы будем считывать из входного файла порции данных фиксированного размера, вызывать функцию преобразования и записывать результат в выходной файл. Хотя мы читаем фиксированные порции данных, их размер не критичен для нашей программы – мы всего лишь оперируем последовательностями символов. Мы можем увеличить или уменьшить размер порций, но программа будет работать правильно.

Первая часть цикла – чтение данных, для которого используется системный вызов *read*. Его поведение было в целом описано выше. Он возвращает количество реально прочитанных байтов или конец файла (число 0).

После чтения блока содержимое регистра %*eax* проверяется на наличие маркера конца файла. Если он найден, то цикл завершается. В противном случае, он продолжается.

После считывания данных вызывается функция *convert_to_upper* с только что прочитанным буфером и количеством символов, прочитанных предыдущим системным вызовом. После того как функция выполнилась, буфер должен быть изменен нужным образом и готов к записи. Затем регистры восстанавливаются сохраненными в стеке значениями.

Аналогично работает системный вызов *write*. Отличие от *read*: копирует данные из буфера в файл. И остается перейти в начало файла.

После выхода из цикла наша программа просто закрывает файловые дескрипторы и завершает работу. Системный вызов *close* требует всего лишь указания дескриптора закрываемого файла в регистре %*ebx*.

Наша программа не идеальна, но ее вполне можно использовать в качестве заготовки при разработке своих программ.

13.8 Работа со структурами данных

Наша программа *toupper* из предыдущего параграфа работала с неструктурированными данными. Она всего лишь обрабатывала текстовые файлы, содержимое которых было кем-то введено. Она не могла интерпретировать данные, поскольку введенный пользователем текст мог быть совершенно случайным.

Структурированные данные – это данные, разделенные на структуры и поля, как правило, фиксированного размера. В связи с этим появляется возможность интерпретации

этих данных. Структурированные данные могут содержать и поля переменной длины, но в этом случае обычно предпочитают работать с базами данных.

Мы попробуем написать программу для манипуляции простыми структурами фиксированной длины. Предположим, мы хотим сохранять информацию о знакомых нам людях. Мы могли бы представить следующий пример структуры фиксированной длины (СЛАЙД 70):

- Имя – 40 байт
- Фамилия – 40 байт
- Адрес – 240 байт
- Возраст - 4 байта

Здесь почти все является символьными данными, за исключением возраста, который является числовым полем, поэтому мы используем 4-байтное длинное слово (мы могли бы ограничиться одним байтом, но длинные слова проще обрабатывать).

В наших последующих программах понадобится доступ к различным частям структуры. Это значит, нам нужно знать смещение каждого поля по отношению к началу структуры. У нас есть несколько констант, описывающих значения таких смещений, и мы их разместим в файле с именем *record-def.s* (СЛАЙД 71).

```
.equ RECORD_FIRSTNAME, 0
.equ RECORD_LASTNAME, 40
.equ RECORD_ADDRESS, 80
.equ RECORD_AGE, 320
.equ RECORD_SIZE, 324
```

К тому же, у нас есть несколько полезных констант (СЛАЙД 72), которые мы можем разместить в файле с именем *linux.s*.

Мы напишем три программы для структуры, описанной в *record-def.s*. Первая программа будет создавать и заполнять файл несколькими структурами. Вторая программа будет отображать структуры в файле. Третья – будет прибавлять 1 год к полю возраста в каждой структуре.

В дополнение к стандартным константам, есть также две функции, которые мы будем использовать в нескольких программах. Первая предназначена для считывания структуры, а вторая – для записи.

В каких параметрах нуждаются эти функции? Нам обычно нужны:

- Адрес буфера, в который считывается структура
- Дескриптор файла, из которого считываются или куда записываются структуры.

Сначала взглянем на нашу функцию чтения (СЛАЙДЫ 73 и 74), которую мы разместим в файле *read-record.s*.

Как видно, это очень простая функция, которая считывает из файла в буфер, размер которого равен длине нашей структуры данных. Функция записи аналогична (СЛАЙДЫ 75 и 76), она расположена в файле *write-record.s*.

Теперь можно приступить к написанию программ.

Запись структур

Первая программа просто пишет во внешний файл несколько структур, содержимое которых задано в коде. Она будет:

- Открывать файл.
- Записывать три структуры.
- Закрывать файл.

Программу, код которой приведен на СЛАЙДАХ 77-80, мы разместим в файле,

названном *write-records.s*.

Сборка и запуск программы достаточно просты.

```
$as write-record.s -o write-record.o
$as write-records.s -o write-records.o
$ld write-records.o write-record.o -o write-records
$./write-records
```

Это приведет к созданию файла *test.dat*, который будет содержать заданные структуры. Тем не менее, поскольку они содержат непечатаемые символы (в частности 0-символ), то в текстовом редакторе не всегда удастся посмотреть содержимое файла. Как не трудно догадаться наша следующая программа будет читать структуры.

Чтение структур

Рассмотрим процесс чтения каждой записи и отображения поля имени каждой структуры. Поскольку у людей имена могут иметь разную длину, то нам понадобится функция, которая будет подсчитывать символы, которые мы хотим вывести на экран. Т.к. мы заполняем каждое поле 0-символами, то мы просто подсчитываем символы до тех пор, пока не достигнем маркера конца строки. Важно, чтобы наши структуры содержали как минимум один такой символ.

На СЛАЙДАХ 81-82 приведен код. Поместим его в файле с именем *count-chars.s*.

Как видно, это довольно простая функция, содержащая цикл с побайтовым просмотром до достижения 0-символа. Подсчитанное количество возвращается вызывающей стороне.

Наша программа *read-records* также не сильно сложная. Она будет делать следующее:

- Открывать файл
- Пытаться прочитать структуру
- Если достигнут конец файла, завершать работу
- В противном случае, она будет считать символы поля имени
- Выводить имя на STDOUT
- Выводить символ новой строки на STDOUT
- Переходить к чтению следующей структуры

Чтобы вывести это, нам понадобится еще более простая функция вывода символа новой строки на STDOUT. Поместим показанный на СЛАЙДЕ 83 код в файл *write-newline.s*.

Теперь все готово к написанию главной программы. Ее код (СЛАЙДЫ 84-86) в файле *read-records.s*.

Сборка и запуск программы достаточно просты.

```
$as read-record.s -o read-record.o
$as read-records.s -o read-records.o
$as count-chars.s -o count-chars.o
$as write-newline.s -o write-newline.o
$ld read-records.o read-record.o write-newline.o \
count-chars.o –o read-records
$./read-records
```

Изменение структур

Теперь мы напишем программу, которая:

- Открывает входной и выходной файлы
- Читает структуры из входного файла
- Инкрементирует возраст

Версия 0.91 pre-release от 07.11.2016. Возможны незначительные изменения.

• Пишет измененную структуру в выходной файл

Как и с двумя предыдущими программами, этот процесс не должен составить большого труда. Код этой программы представлен на СЛАЙДАХ 87-89. Файл назовем *add-year.s*.

Сборка и запуск программы достаточно просты.

```
$as read-record.s -o read-record.o
$as write-record.s -o write-record.o
$as add-year.s -o add-year.o
$ld read-record.o write-record.o add-year.o -o add-year
$./add-year
```

Эта программа прибавит один год к каждой структуре в файле *test.dat* и запишет результатт в файл *testout.dat*.

Как видно, работа со структурами фиксированной длины предельно проста. Мы читаем блоки данных в буфер, обрабатываем их и записываем обратно. К сожалению, наша программа не выводит обновленное значение возраста на экран, так что мы не можем сразу же удостовериться в корректности и эффективности нашей программы. Студентам предлагается сделать это в качестве самостоятельного упражнения.

Литература и дополнительные источники к Теме 13

1. Магда, Ю.С. Ассемблер для процессоров Intel Pentium/ Ю.С. Магда. – СПб.: Питер, 2006. – 416 с.
2. Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
3. Столяров, А.В. Программирование на языке ассемблера NASM для ОС UNIX: учеб.пособие. – М.: Макс, 2011. – 188 с. – Доступ: http://www.stolyarov.info/books/asm_unix
4. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA.
5. Intel® 64 and IA-32 Architectures Software Developer’s Manual, 4.1 Instructions (N-Z), PUSH