

GCC Calling conventions investigation

© Андрей Чесноков, май 2011

Содержание

[Ссылки](#)

[Предисловие](#)

[Компиляция исходников в ассемблер](#)

[Обычные соглашения C о вызове функций](#)

[Оптимизация](#)

[Исследование объектных файлов](#)

[Изменение соглашений передачи параметров](#)

[Передача параметров через регистры](#)

[Итоги](#)

Ссылки

<http://www.wikihow.com/Make-a-Computer-Operating-System> - HowTO make OS

<http://www.linuxfromscratch.org/lfs/> - Linux from Scratch Project

<http://www.brokenthorn.com/Resources/> Некая мелкая компания, разрабатывающая свою ОС. Всего 2 человека, судя по описанию.

http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/Chapter_20/CH20-2.html -

Keyboard Controller Programming in the book The art of Assembly language

http://www.codepedia.com/1/x86ASMFAQ_Hardware

<http://tldp.org/LDP/LG/issue77/krishnakumar.html> Writing Your Own Toy OS (Part I)

<http://tldp.org/LDP/LGNET/79/krishnakumar.html> Writing Your Own Toy OS (Part II)

<http://tldp.org/LDP/LGNET/82/raghu.html> Writing Your Own Toy OS (Part III)

<http://maketecheasier.com/books/> Книжки по Линуксу в том числе Building Linux from Scratch

<http://www.osdever.net/tutorials/index> - Множество ссылок на полезные руководства

http://ru.wikipedia.org/wiki/Hardware_abstraction_layer Hardware Abstraction Layer (HAL)

http://www.jamesmolloy.co.uk/tutorial_html/index.html

<ftp://ftp.lucky.net/pub/Linux/mirrors/deepstyle.org.ua/downloads/deepstyle-current/extra2-ds/Documentation/gazette.linux.ru.net/rus/articles/toy-os/toy-os.html> Статья по мотивам Индийской Toy OS

<http://www.osdever.net/FreeVGA/vga/crtcreg.htm> VGA programming
<http://www.intel-assembler.it/portale/5/VGA-Programmers-Master-Reference-Manual/VGA-Programmers-Master-Reference-Manual.asp> VGA Programmers Reference Manual
<http://www.intel-assembler.it/portale/indice.asp?sz=100> Hardware Manuals
<http://www.gamedev.ru/code/forum/?id=17661&page=2> memcpy asm
<http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/Function-Attributes.html#Function-Attributes>

<http://bellard.org/tcc/> Tiny CC

Предисловие

В предыдущих экспериментах с Boot Loader-ом для операционной системы сделан Boot Loader на основе Boot Loader от Menuet OS. Он загружает ядро по адресу 0x10000, переводит CPU в защищенный режим и передает на ядро управление. Открылось поле для дальнейших экспериментов.

Wiki "how to" (<http://www.wikihow.com/Make-a-Computer-Operating-System>) советует после создания Boot Loader-а сосредоточиться на выводе текста и обработке прерываний, написании библиотеки для себя, для отладки. Согласен с этим. Добавлю лишь, что хочется побольше применять готового кода, а не разрабатывать всё с нуля. Это бесперспективно. Для этого нужно из писателя переквалифицироваться в читателя. Я не даром эту статью начал со множества ссылок. Чтобы что-то позаимствовать у предшественников, нужно уметь читать код.

Поэтому для начала несколько ссылок по gcc и GNU assembler.

http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/i386-and-x86_002d64-Options.html#i386-and-x86_002d64-Options Опции компилятора.
<http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Function-Attributes.html#Function-Attributes> gcc function attributes.
<http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html> GCC Inline assembly
<http://sourceware.org/binutils/docs-2.21/as/index.html> GNU Assembler

Теперь подробнее. Перехожу к экспериментам. Беру такой вот main.c.

```
#include <stdlib.h>
```

```
int func(int a, const char *b)
{
    return atoi(b)+a;
}
```

```
int main()
{
    return func(1,"1");
}
```

Пример ничего не значащий. Я собираюсь применять его только для изучения ассемблерных соглашений вызова gcc. Т.н. calling conventions. Кроме того хочу посмотреть во что gcc компилирует код.

Компиляция исходников в ассемблер

Компилирую приведенный выше исходник в ассемблер.

```
$ gcc -S main.c
```

Опция -S предписывает gcc скомпилировать исходный код вместо объектного файла в ассемблерный файл. Полученный ассемблерный файл ниже. Прежде всего бросается в глаза необычный синтаксис ассемблера. Это синтаксис AT&T. Он отличается от привычного Intel-ового синтаксиса порядком операндов в операциях.

```
.file    "main.c"
.text
.p2align 2,,3          ; выравнивание функции в памяти
.globl func
.type    func, @function ; необязательная декларация
func:    ; метка, определяющая начало функции func
    pushl %ebp          ; сохраняется в стеке регистр ebp (функция
                        ; не имеет права его менять
    movl  %esp, %ebp     ; ebp = esp
    subl  $8, %esp       ; esp -= 8
    subl  $12, %esp      ; esp -=12
    pushl 12(%ebp)       ; помещается в стек второй параметр функции (ebp+12)
    call  atoi           ; вызывается atoi
    addl  $16, %esp       ; esp +=16
    addl  8(%ebp), %eax   ; eax += a (первый параметр функции a)
    leave                ; mov esp, ebp; pop ebp
    ret                 ; return
.size    func, .-func
.section .rodata
.LC0:
.string  "1"
.text
.p2align 2,,3
.globl main
```

```

.type    main, @function
main:
    pushl %ebp
    movl  %esp, %ebp    ; стандартный вход в функцию
    subl  $8, %esp      ;
    andl  $-16, %esp    ; выравнивание указателя стека на 16-байтную границу
    movl  $0, %eax      ; eax = 0
    addl  $15, %eax     ; eax +=15
    addl  $15, %eax     ; eax +=15
    shrl  $4, %eax      ; eax >>=4
    sall  $4, %eax      ; ??
    subl  %eax, %esp    ; esp -= eax
    subl  $8, %esp      ; esp -= 8
    pushl $.LC0         ; push (char *)"1"
    pushl $1            ; push 1
    call  func          ; вызов функции
    addl  $16, %esp     ; восстановление esp
    leave
    ret
.size    main, .-main
.ident   "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"

```

Например в Intel-овом синтаксисе:

```
mov eax, ebx    ; присвоение регистру eax значения из регистра ebx
```

В AT&T порядок операндов строго обратный:

```
mov %ebx, %eax ; присвоение регистру eax значения из регистра ebx
```

Кроме того перед операндами ставятся необычные префиксы. Перед названиями регистров %. Перед целочисленными константами \$.

В остальном ассемблер вполне обычный. Теперь о соглашениях о вызовах функций.

Обычные соглашения C о вызове функций

Соглашения определяют способ передачи параметров в функции, способ возврата “возвращаемого значения”, соглашения о том, какие регистры имеет право менять функция, какие не имеет. Обычные соглашения C о передаче параметров заключаются в следующем.

1. Параметры в функцию передаются через стек. Вызывающая функция помещает параметры в стек в обратном порядке, начиная с последнего к первому. Фрагмент рассмотренного кода это подтверждает:

```

        .section      .rodata
.LC0:
        .string "1"

        pushl  $.LC0      ; push (char *)"1"
        pushl  $1         ; push 1
        call   func       ; вызов функции
        addl   $16, %esp   ; восстановление esp

```

В стек помещаются второй аргумент - указатель на строчку в сегменте `.rodata`. Потом помещается первый аргумент - единица. И функция вызывается. После вызова, вызывающая функция сама заботится о стеке увеличивает указатель стека на размер положенных в него данных. Про это надо помнить, если производишь вызов функций из стандартных библиотек на ассемблере.

2. Возвращаемое значение помещается функцией в регистр `eax`. Если оно `float point`, то через `ST0`.

3. Функция на C предохраняет от изменения внутри себя все регистры, кроме `eax`, `edx` и `ecx`.

Энциклопедические сведения про соглашения о передаче параметров доступны в wiki:
http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

The **cdecl** calling convention is used by many C systems for the x86 architecture.[1] In cdecl, function parameters are pushed on the stack in a right-to-left order. Function return values are returned in the EAX register (except for floating point values, which are returned in the x87 register ST0). Registers EAX, ECX, and EDX are available for use in the function.

Еще раз посмотрим на скомпилированный код.

```

        subl   $8, %esp      ; esp -= 8
        subl   $12, %esp     ; esp -=12
        pushl  12(%ebp)      ; помещается в стек второй параметр функции
        call   atoi         ; вызывается atoi
        addl   $16, %esp     ; esp +=16

```

Что странного в вышеприведенном коде? Наличие странных операций с регистром стека.

Делается это ради выравнивания указателя стека на 16-ти байтную границу. Делается это из-за оптимизации для работы с sse инструкциями для манипуляции с вещественными переменными в стеке... (Возможно просто для хранения возвращаемых адресов функций на 16-ти байтных границах). Почему встречаются повторяющиеся инструкции? Например в функции func указатель стека вычитается дважды.

Оптимизация

Это удивляет. Но это результат работы компилятора без оптимизации. Если повторить ту же процедуру с опцией -O2, то инструкция заменится на одну. Ниже код скомпилированный командой

```
$ gcc -O2 -S main.c
```

Сразу видно, что код стал много короче. Вывод: опция O2 является просто жизненно необходимой.

```
.file    "main.c"
.text
.p2align 2,,3
.globl func
.type   func, @function
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $20, %esp
    pushl   12(%ebp)
    call    atoi
    addl    8(%ebp), %eax
    leave
    ret
.size     func, .-func
.section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "1"
    .text
    .p2align 2,,3
.globl main
.type     main, @function
main:
    pushl   %ebp
```

```

movl   %esp, %ebp
subl   $8, %esp
andl   $-16, %esp
subl   $24, %esp
pushl  $.LC0
pushl  $1
call   func
leave
ret
.size   main, .-main
.ident  "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"

```

Исследование объектных файлов

Объектные файлы можно обследовать утилитой `objdump`. Она входит в комплект `binutils`, поставляемых вместе с `gcc`. Чтобы поэкспериментировать, компилирую, сначала свой пример в объектный файл (флаг `-c` компилировать в объектный файл, `-O2` чтобы было меньше кода:

```
$ gcc -O2 -c main.c
```

Теперь полученный файл диассемблирую с помощью `objdump`:

```
$ objdump -r -d main.o
```

(`-d` деассемблировать `-r` показывать внешние ссылки в виде имен)

```
main.o:      file format elf32-i386-freebsd
```

Disassembly of section `.text`:

```
00000000 <func>:
```

```

0:  55          push  %ebp
1:  89 e5       mov   %esp,%ebp
3:  83 ec 14    sub   $0x14,%esp
6:  ff 75 0c    pushl 0xc(%ebp)
9:  e8 fc ff ff call  a <func+0xa>
               a: R_386_PC32      atoi
e:  03 45 08    add   0x8(%ebp),%eax
11: c9         leave
12: c3         ret
13: 90         nop

```

```
00000014 <main>:
```

```
14: 55          push  %ebp
```

```

15: 89 e5          mov  %esp,%ebp
17: 83 ec 08       sub  $0x8,%esp
1a: 83 e4 f0       and  $0xffffffff,%esp
1d: 83 ec 18       sub  $0x18,%esp
20: 68 00 00 00 00 push  $0x0
                21: R_386_32 .rodata.str1.1
25: 6a 01          push $0x1
27: e8 fc ff ff    call 28 <main+0x14>
                28: R_386_PC32 func
2c: c9            leave
2d: c3            ret

```

Здесь видно смещение каждой инструкции, инструкции в виде шестнадцатиричных кодов и в виде ассемблерных команд. Все наглядно.

У objdump есть дополнительные опции, например -h вывести информацию о секциях внутри объектного файла.

```
$ objdump -h main.o
```

```
main.o: file format elf32-i386-freebsd
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000002e	00000000	00000000	00000034	2**2
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.data	00000000	00000000	00000000	00000064	2**2
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000000	00000000	00000000	00000064	2**2
	ALLOC					
3	.rodata.str1.1	00000002	00000000	00000000	00000064	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.comment	00000025	00000000	00000000	00000066	2**0
	CONTENTS, READONLY					

Здесь можно увидеть размер каждой секции, смещение внутри объектного файла, выравнивание, VMA, LMA - адреса (это адреса, по которым секция будет загружаться в целевой выполнимый файл и в память). Подробности про VMA и LMA можно посмотреть в документации на линкер: <http://sourceware.org/binutils/docs/ld/>

С помощью objdump можно смотреть ещё много чего интересного. Например содержимое библиотек. Подробности смотреть в его описании. Мне пока для информационно-отладочных нужд достаточно опций -h -r -d.

Ещё одна полезная утилита, помогающая разбираться с объектными файлами: nm. Она выводит глобальные имена в объектном файле или библиотеке. Т.е. имена функций и глобальных переменных. Например:

```
$ nm main.o
```

```
U atoi
00000000 T func
00000014 T main
```

Здесь важно что. Здесь видно, что функция atoi - внешняя ссылка, её надо искать в другом объектном файле. Это определяется значком "U" перед именем функции. "T" перед именами func и main означает, что это имена функций скомпилированный код которых лежит в этом объектном файле. Эти имена функций будут видны линкеру.

Изменение соглашений передачи параметров

gcc, как и многие другие компиляторы, позволяет модифицировать соглашение по передаче параметров в функции. Это возможно разными способами. Во первых это можно сделать с помощью командной строки. Например -mrtcd позволяет указать, что функции с фиксированным количеством аргументов будут сами вычищать за собой стек с помощью инструкции "ret num". Это избавит от необходимости лишних операций со стеком в вызывающей функции:

```
pushl $.LC0      ; push (char *)"1"
pushl $1         ; push 1
call    func     ; вызов функции
;добавление к esp 8-ми в этом месте не требуется
```

Это может сделать код более читаемым. Но недостаток опции компилятора -mrtcd заключается в том, что действовать она будет на все вызовы компилятора. Компилятор будет считать, что все функции, в том числе во всех библиотеках скомпилированы с этой опцией. А это как правило не так (по крайней мере в UNIX-like системах).

Чтобы была возможность изменять порядок передачи параметров выборочно у отдельных функций, применяется специальная синтаксическая конструкция "Function Attributes" (<http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Function-Attributes.html#Function-Attributes>).

Выглядит конструкция, как показано в примере ниже. Например, атрибут stdcall определяет для отдельной функции такие же соглашения, как опция -mrtcd компилятора.

```
#include <stdlib.h>
```

```
__attribute__((stdcall)) int func(int a, const char *b)
```

```
{
    return atoi(b)+a;
}
```

```
int main()
{
    return func(1,"1");
}
```

Компилируем:

```
$ gcc -O2 -S main.c
```

Смотрим результат:

```
.file    "main.c"
.text
.p2align 2,,3
.globl func
.type    func, @function
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $20, %esp
    pushl   12(%ebp)
    call    atoi
    addl    8(%ebp), %eax
    leave
    ret     $8
.size     func, .-func
.section   .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "1"
    .text
    .p2align 2,,3
.globl main
.type     main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    subl    $24, %esp
    pushl   $.LC0
```

```

pushl $1
call func
leave
ret
.size main, .-main
.ident "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"

```

Видно, что функция `func` завершается инструкцией `ret $8`, выбрасывающей из стека 8 байт. В данном случае на вызывающую функцию `main` это никак не повлияло. После вызова `func` она завершается последовательностью `leave, ret`. Так же было и в обычном оптимизированном варианте.

Передача параметров через регистры

Атрибут `regparm (number)` указывает компилятору, что первые (до 3-х) параметры передаются через стек соответственно в `eax`, `edx`, `ecx`.

Документация говорит:

Beware that on some ELF systems this attribute is unsuitable for global functions in shared libraries with lazy binding (which is the default). Lazy binding will send the first call via resolving code in the loader, which might assume `EAX`, `EDX` and `ECX` can be clobbered, as per the standard calling conventions. Solaris 8 is affected by this. GNU systems with GLIBC 2.1 or higher, and FreeBSD, are believed to be safe since the loaders there save `EAX`, `EDX` and `ECX`. (Lazy binding can be disabled with the linker or the loader if desired, to avoid the problem.)

Т.е. на некоторых системах (конкретно на Солярисе) с этой опцией проблемы при использовании для глобальных функций в динамических библиотеках.

В моём случае я не собираюсь пока использовать динамических библиотек вообще. Так что меня может такой способ передачи параметров вполне устроить.

```

__attribute__((regparm(2))) int func(int a, const char *b)
{
    return atoi(b)+a;
}

int main()
{
    return func(1,"1");
}

```

Компилирую gcc -O2 -S main.c, получаю

```
.file "main.c"
.text
.p2align 2,,3
.globl func
.type func, @function
func:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $16, %esp
    pushl %edx
    movl %eax, %ebx
    call atoi
    addl %ebx, %eax
    movl -4(%ebp), %ebx
    leave
    ret
.size func, .-func
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "1"
    .text
    .p2align 2,,3
.globl main
.type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, %edx
    movl $1, %eax
    call func
    leave
    ret
.size main, .-main
.ident "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"
```

Анализируем полученный код. В функции main вызов func выглядит так:

```
movl  $.LC0, %edx
movl  $1, %eax
call  func
```

Логично. Параметры передаются через регистры. Но самое интересное для анализа в реализации func. В её реализации видно, что регистр ebx предохраняется компилятором.

func:

```
pushl  %ebp
movl   %esp, %ebp
pushl  %ebx          ; ebx сохранен в стеке
subl   $16, %esp
pushl  %edx          ; второй параметр, переданный в func помещается в стек
movl   %eax, %ebx    ; первый параметр сохранился в ebx
call   atoi
addl   %ebx, %eax     ; вот это самое интересное, компилятор считает что ebx
                    ; не изменяется внутри atoi. (и это правильно).
movl   -4(%ebp), %ebx ; ebx восстановлен из стека
leave
ret
```

Т.е. как говорилось выше, по соглашению о регистрах функция не восстанавливает только содержимое регистров EAX, EDX, ECX. Остальные регистры в случае изменения должны восстанавливаться в первоначальный вид перед завершением функции.

Итоги

- В приведенных примерах рассмотрены соглашения о передаче параметров в gcc. Рассмотрены несколько параметров командной строки gcc.
- Описан синтаксис полезной конструкции `__attribute__((...))`.
- Описаны пара возможностей по изменению соглашения о передаче параметров, в частности, описана передача параметров через регистры.
- Полный список возможных атрибутов, которые можно использовать в исходном коде смотреть здесь:
<http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Function-Attributes.html#Function-Attributes>
- Список параметров командной строки проще всего смотреть здесь:
http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/i386-and-x86_002d64-Options.html#i386-and-x86_002d64-Options