КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 13 Взаимодействие С — ассемблер

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

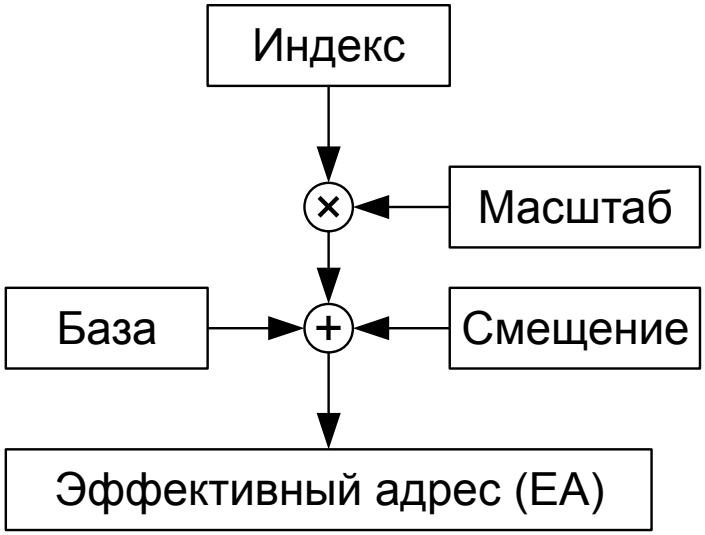
2022.04.20

Оглавление

Адресация памяти в x86 (IA32)

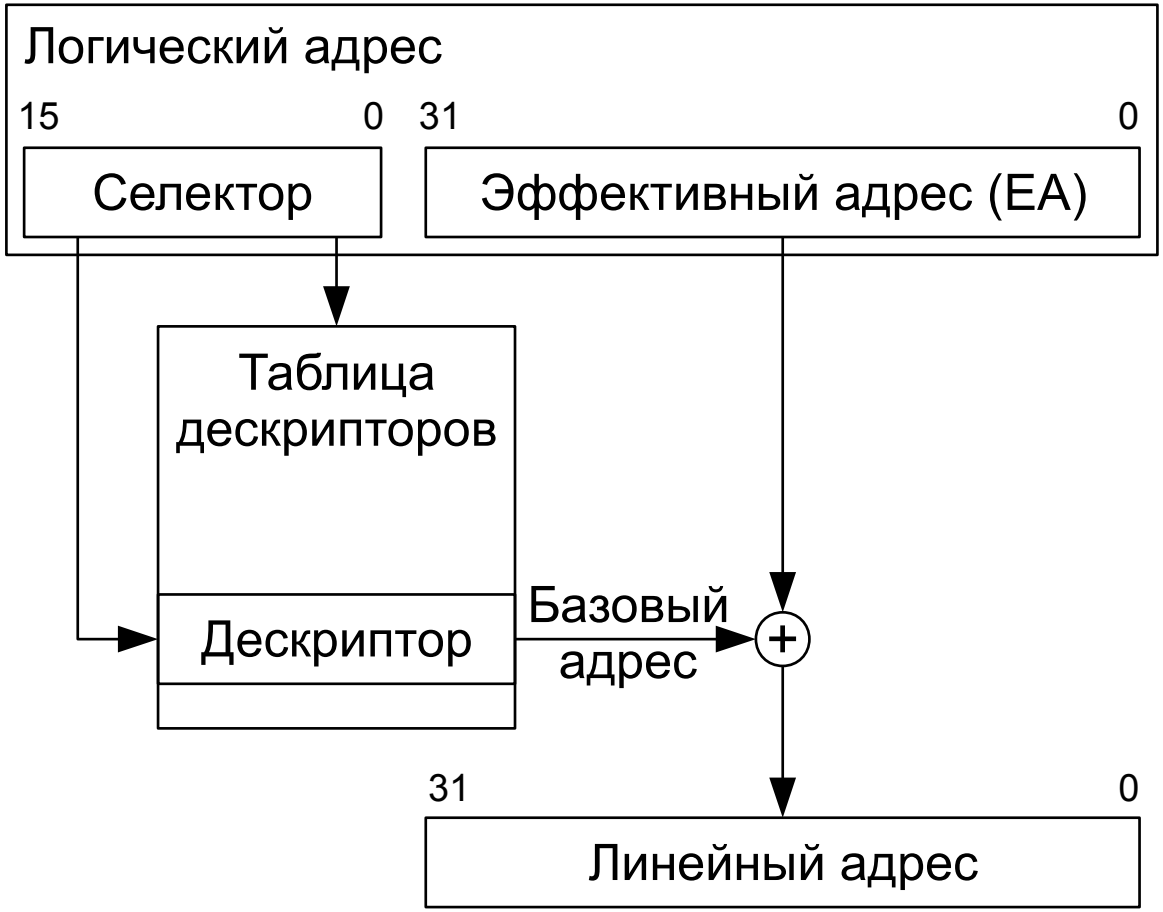
Существует четыре вида адресов:
1. эффективный (адрес относительно начала сегмента) — кодируется в инструкции;
2. логический (сегмент и эффективный адрес);
3. линейный (или виртуальный) — формируется в процессе сегментации;
4. физический — в системной памяти — процесс страничной трансляции;

Формирование эффективного адреса



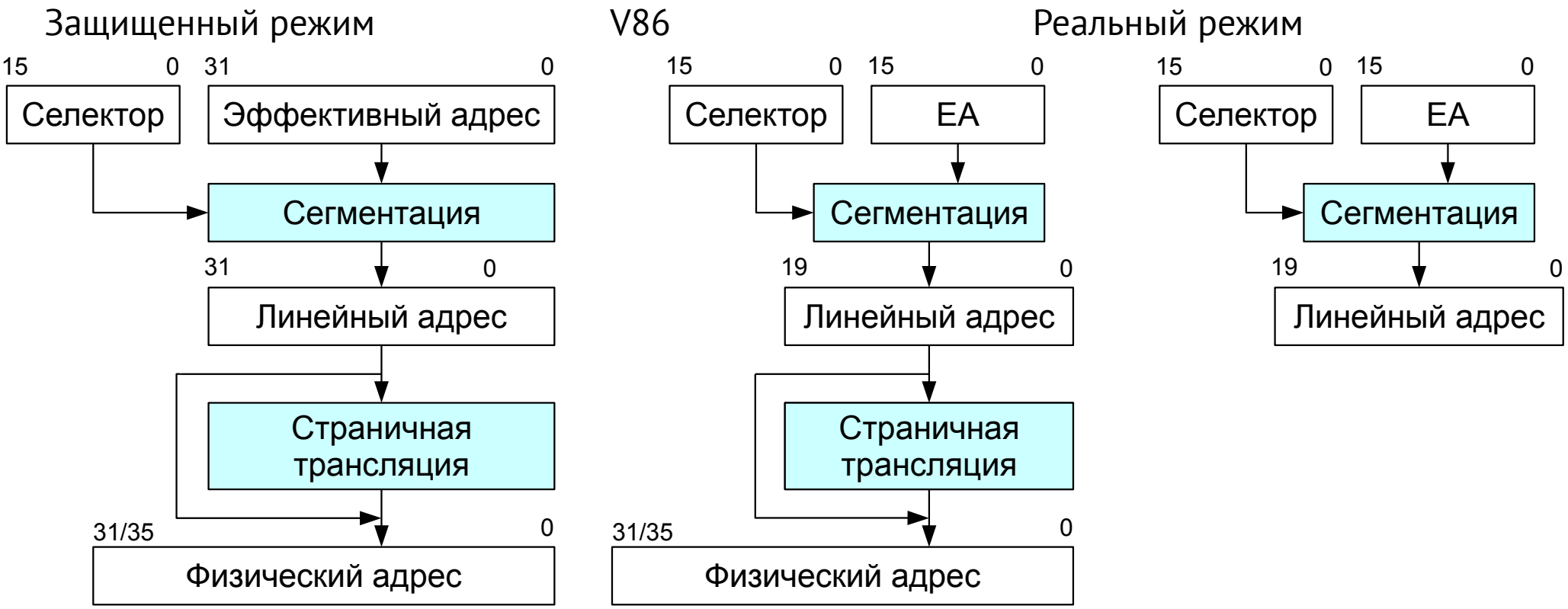
База — может содержаться в любом из РОН;
Индекс — может содержаться в любом из РОН за исключением ESP;
Смещение — содержится в коде команды;
Масштаб — содержится в коде команды (1, 2, 4, 8).

Формирование линейного адреса (сегментация)



1. из селектора извлекается поле индекса;
2. по индексу находится соответствующий дескриптор;
3. из дескриптора извлекается поле адреса базы.
4. К адресу базы добавляется смещение.

Формирование физического адреса



Взаимодействие с 32-разрядным кодом

Нижеизложенное касается для *UX, Win32 и DJGPP (DJ's GNU Programming Platform — проект по переносу GNU-утилит на платформы DOS и Windows для поддержки DPMI).

Ниже описаны некоторые распространенные вопросы создания ассемблерного кода для взаимодействия с 32-битными подпрограммами C и как писать позиционно-независимый код для совместно используемых библиотек.

Почти весь 32-битный код, и в частности весь код, работающий под Win32, DJGPP или любым из вариантов ПК *UX, работает в модели с плоской памятью.

Это означает, что регистры сегментов и страничная подсистема уже настроены для предоставления приложению одинакового 32-битного адресного пространства размером 4 Гб независимо от сегмента, в связи с чем **следует полностью игнорировать все регистры сегментов**.

При написании кода приложений с плоской моделью никогда не нужно ни использовать переопределение сегмента, ни изменять какой-либо из сегментных регистров, поскольку адреса в кодовых секциях, куда передают управление CALL и JMP, находятся в том же адресном пространстве, что и адреса переменных в секциях данных, а также и адреса локальных переменных и параметры процедуры в секциях стека.

Каждый адрес имеет длину 32 бита и содержит только компоненту смещения.

Организация памяти приложения

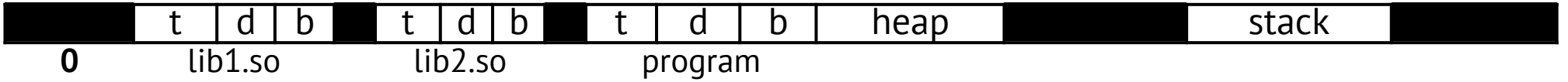
Детальная разбивка памяти существенно зависит как от процессора, так и от операционной системы. Программа для выполнения считывается в память, где и остается вплоть до своего завершения.

Статический код программ считывается в нижнюю часть памяти (по меньшим адресам).

Программе выделяется ряд блоков памяти специального назначения для разных типов данных.

В нижней части адресного пространства, которую можно использовать для стека, может существовать **«защитная» область»** (guard area). Защитная область никогда не отображается на физическую память и если либо стек, либо куча пытаются в него «врасти», немедленно возникает исключение ошибки сегментации.

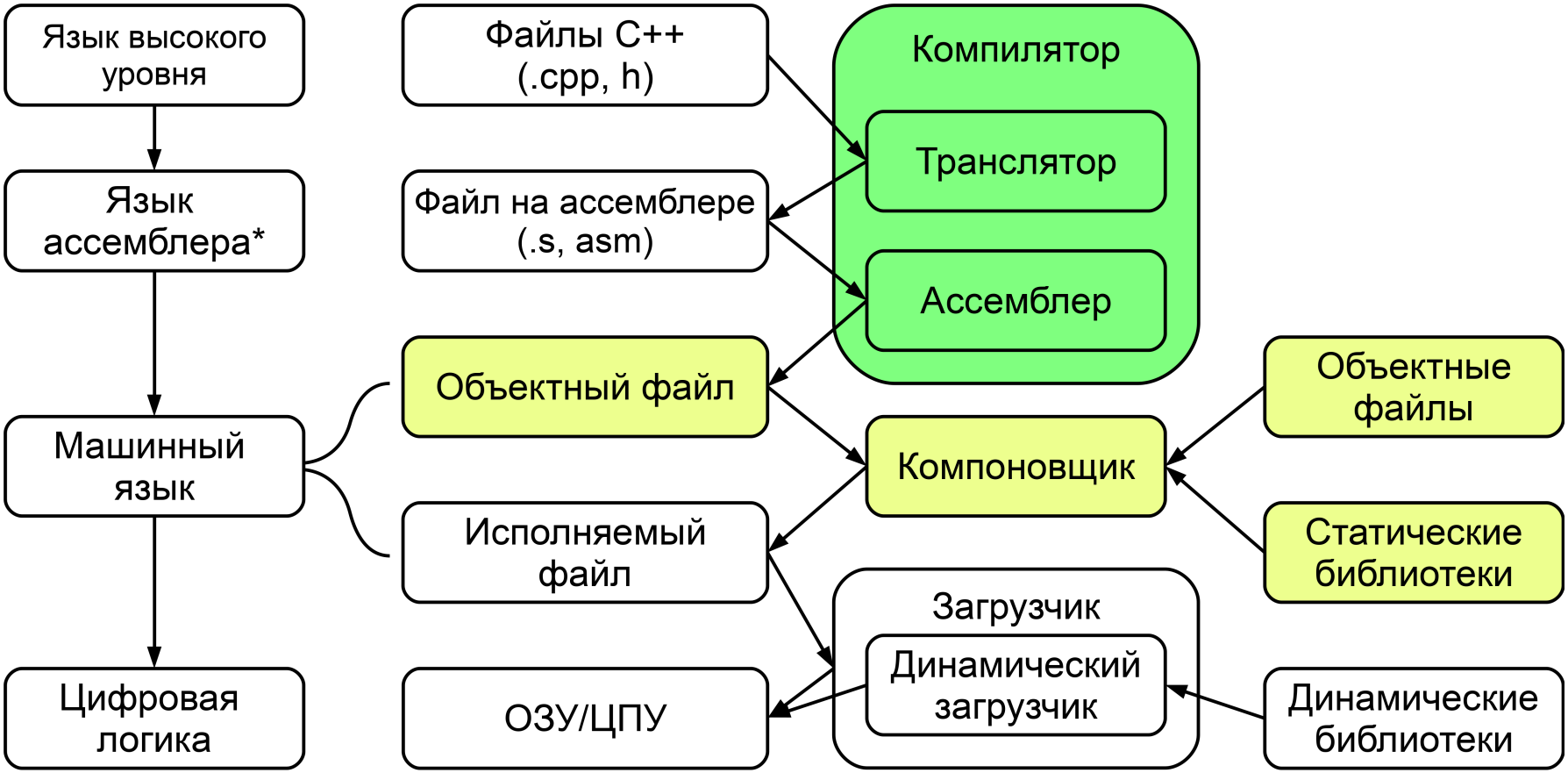
Адресное пространство выглядит следующим образом:



Черные области на этой диаграмме не отображаются на физическую память и любая попытка к ним доступа вызывает немедленную ошибку сегментирования.

Размер таких областей для 64-разрядных программ обычно намного превышает размер отображаемой памяти.

Связь между файлами программы и утилитами



make — утилита, отслеживающая изменения в файлах и вызывающая необходимые программы из набора, использующегося для компиляции и генерации выполняемого кода из исходных текстов (toolchain).

Взаимодействие с 32-битными C-программами

Имена внешних символов

Большинство 32-битных компиляторов C разделяют соглашение, используемое 16-битными компиляторами, что имена всех глобальных символов (функций или данных), которые они определяют, формируются путем добавления префикса подчеркивания к имени, которое написано в программе на C.

```
extern _printf                      ; чтобы asm-программа увидела C-символ printf
%define printf _printf

global _my_asmproc                  ; чтобы C-программа увидела символ  my_asmproc
%define my_asmproc _my_asmproc
```

Однако, не все из них ведут себя именно так. В частности, спецификация ELF гласит, что C-символы не имеют начальных подчеркиваний в именах на ассемблере.

Это означает, что для связывания с символами из кода на C, следует использовать директиву **`extern`** без каких-либо подчеркиваний и прочих украшений.

```
extern printf
global my_asmproc
```

Для формата ELF начальное подчеркивание использоваться не должно

```
$ cat printf.asm
;global   _start
global   main
extern   printf
;                              printf("int value: %d (12345)\n", number)
section .data
number   dd  12345
message  db "int value: %d (12345)", 10, 0

section .text
_start:
main:
         push    dword [number]
         push    dword message
         call    printf
         add     esp, byte 8
;---------------------------------------
         mov     ebx, 0   ; exit code
         mov     eax, 1   ; 1 = sys_exit
         int     0x80
```

```
$ nm printf.o
00000000 T main
00000004 d message
00000000 d number
         U printf
00000000 t _start
```

```
$ cat makefile
NAME=printf

AS=nasm
CC=gcc
#LD=ld
LD=gcc

LDFLAGS=-m32
ASFLAGS=-Wall -f elf -g

.SUFFIXES:
.SUFFIXES: .o .c .asm

all: $(NAME)

$(NAME): $(NAME).o
   $(LD) $(LDFLAGS) $^ -o $@

.PHONY: clean
clean:
   $(RM) $(NAME) *.o *.lst

$(NAME).o: $(NAME).asm makefile
   $(AS) $(ASFLAGS) -l $(*F).lst $< -o $@
```

$^ — имена всех зависимостей, с пробелами между ними;

$@ — имя файла цели правила;

$< — имя первой зависимости.

Соглашение о вызове C в 32-битных программах

1) Вызывающая сторона помещает параметры функции в стек один за другим в обратном порядке (справа налево, так что первый аргумент, указанный для функции, помещается в стек последним).

2) Затем вызывающая сторона, выполняя команду ближнего вызова **CALL**, передает управление вызываемой стороне.

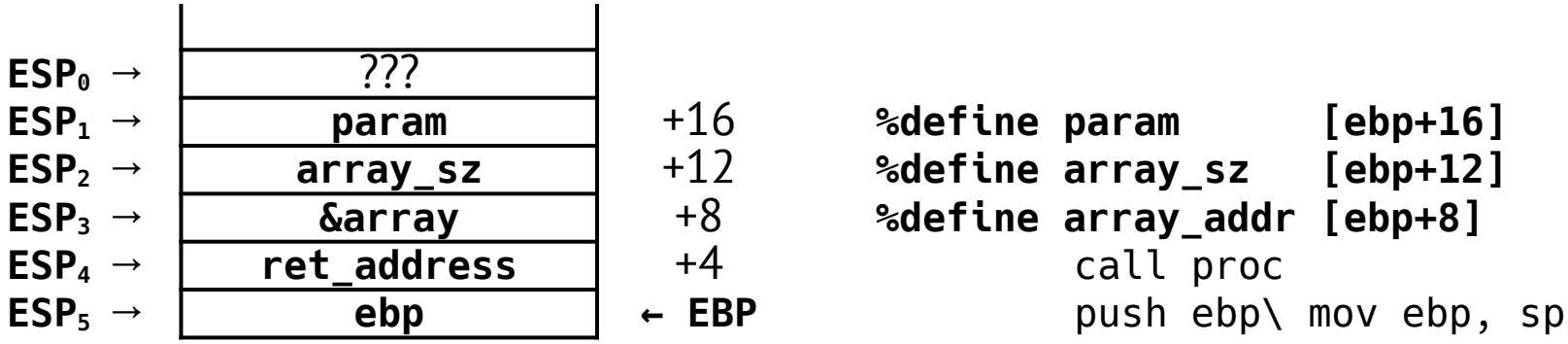
3) Вызываемая сторона получает управление, и обычно (это не является необходимым в случае функций, которым не требуется доступ к их параметрам) начинает работу с сохранения содержимого регистра **ESP** в регистре **EBP**, чтобы иметь возможность использовать **EBP** в качестве указателя базы для получения доступа к параметрам в стеке.

Однако вызывающая сторона, скорее всего, сделала тоже самое, поэтому соглашение о вызовах требует, чтобы **EBP** сохранялось любой C-функцией.

Следовательно, вызываемая сторона, если она собирается использовать **EBP** в качестве указателя кадра, должна сначала сохранить в стеке его предыдущее значение.

```
push ebp
mov  ebp, esp
```

4) После этого вызываемая сторона может получить доступ относительно **EBP** к своим параметрам.



Двойное слово в **[EBP]** содержит предыдущее значение **EBP** в том виде, в котором оно было до сохранения;

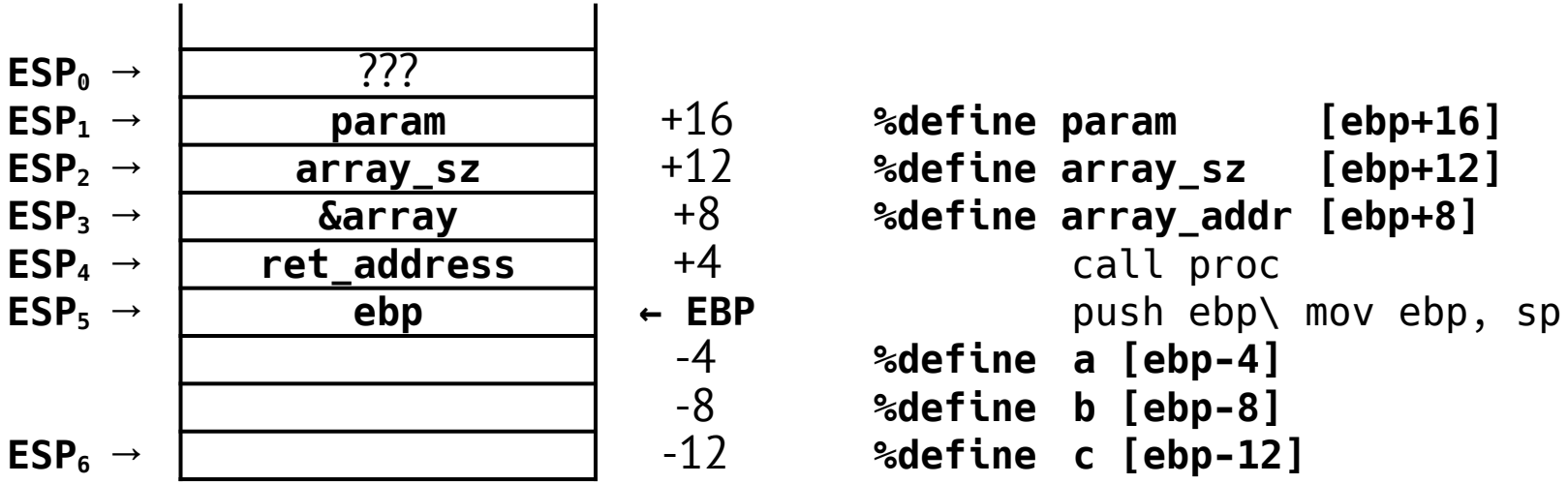
Следующее двойное слово в **[EBP+4]** содержит адрес возврата, неявно сохраненный инструкцией **CALL**.

После них, начиная с **[EBP+8]**, идут параметры — именно по этому смещению относительно **EBP** будет доступен крайний левый параметр функции, поскольку он был сохранен в стеке последним. Остальные, соответственно, располагаются с последовательно увеличивающимися на 4 смещениями.

Таким образом, в функции, например, такой как **printf()**, принимающей переменное число параметров, помещение в стек параметров в обратном порядке позволяет функции знать, где найти свой первый параметр, который сообщает ей количество и тип остальных.

5) Вызываемая сторона может также еще больше уменьшить **ESP**, чтобы выделить в стеке место для локальных переменных. Доступ к ним осуществляется при помощи отрицательных смещений относительно **EBP**.

```
push ebp
mov  ebp, esp
sub  esp, 3*4 ; фрейм под локальные переменные a, b и c
```

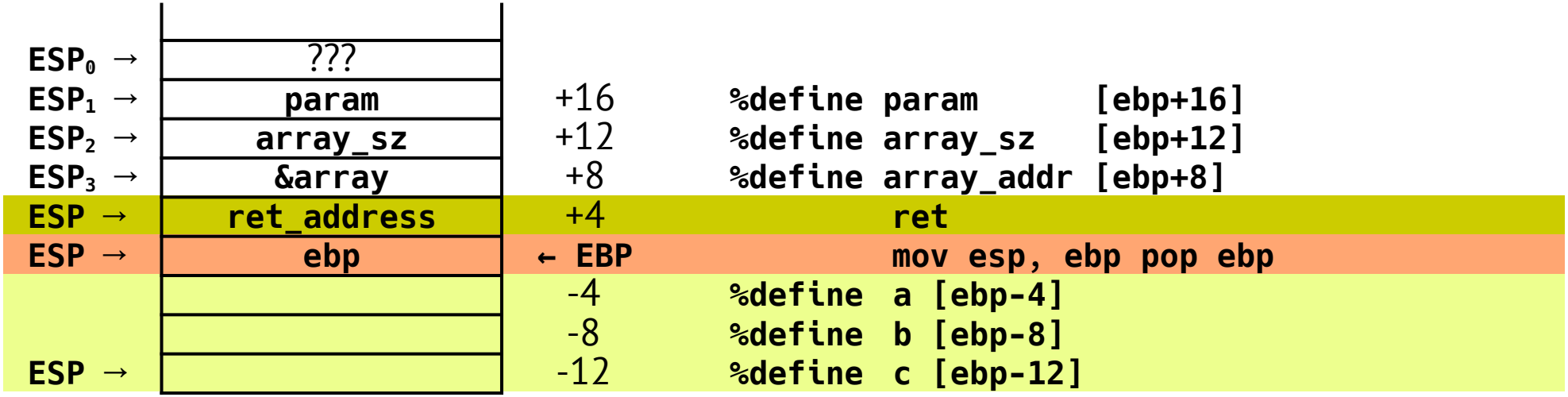


6) Вызываемая сторона, если она желает вернуть значение вызывающей стороне, должна оставить это значение в регистрах **AL**, **AX** или **EAX** в зависимости от размера возвращаемого значения.

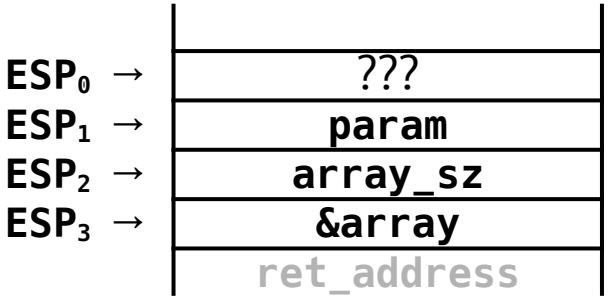
Результаты вычислений с плавающей точкой обычно возвращаются в **ST0**.

7) После того, как вызываемая сторона завершила работу, она восстанавливает **ESP** из **EBP** в случае, если она выделяла пространство в стеке для размещения локальных переменных, после чего извлекает предыдущее значение **EBP** и возвращает через **RET** или **RET N** управление вызывающей стороне.

```
mov esp, ebp
pop ebp
ret
```



8) Когда вызывающая сторона получает управление от вызываемой, параметры функции все еще находятся в стеке, поэтому обычно вызывающая сторона добавляет к **ESP** непосредственную константу для их удаления (вместо выполнения ряда медленных инструкций **POP**). Таким образом, если функция случайно была вызвана с неправильным количеством параметров из-за несоответствия прототипа, стек все равно будет возвращен в правильное состояние, поскольку удаление параметров выполняет вызывающая программа, которая знает, сколько параметров она туда поместила.




```
add esp, 12 ; caller does it
```

Соглашение о вызовах API Windows для программ Win32

Существует альтернативное соглашение о вызовах, используемое программами Win32 для вызовов API Windows, а также для функций, вызываемых API Windows, таких как оконные процедуры.

Все они следуют соглашениям, которые Microsoft называет соглашением **`__stdcall`**.

- стек очищает вызываемая сторона, передавая необходимое значение в инструкцию **`RET N`**;
- параметры в стек помещаются в порядке справа налево;
- перед именем ставится символ подчеркивания '_';
- после имени ставится символ '@', за которым идет в десятичном выражении количество байт в списке аргументов.

Таким образом, функция, объявленная как **`int func(int a, double b)`** будет декорироваться как **`_func@12`**.

Пример функции в стиле C

```
global  _myfunc

_myfunc:
        push   ebp
        mov    ebp, esp
        sub    esp, 0x40    ; 64 байта для локальных переменных
        mov    ebx, [ebp+8] ; первый параметр, переданный функции

        ; какой-то код

        leave               ; mov esp, ebp\ pop ebp
        ret
```

Где-то в другом месте процесса, чтобы вызвать C-функцию из ассемблерного кода, необходимо сделать что-то вроде следующего:

```
extern  _printf

section .text

        ; какой-то код ...

        push    dword [number] ; целочисленная переменная (аргумент)
        push    dword message  ; указатель в сегмент данных (строка формата)
        call    _printf
        add     esp, byte 8    ; `byte' укорачивает поле imm в коде инструкции

        ; какой-то код ...

section .data

number     dd   1234
message    db   'Это число -> %d <- должно быть 1234',10,0
```

Данный фрагмент кода является ассемблерным эквивалентом кода на C:

```
int number = 1234;
printf("Это число -> %d <- должно быть 1234\n", number);
```

Полный пример вызова `printf()` из кода на ассемблере

```
global   main    ; компоновка gcc требует точку старта с именем "main"
extern   printf  ; в libc

section .data
number   dd  12345
message  db "int value: %d (12345)", 10, 0

section .text
main:
         push    dword [number]
         push    dword message
         call    printf          ; вызов в libc
         add     esp, byte 8     ; подбираем стек
;--- exit() ---------------------------------
         mov     ebx, 0   ; exit code
         mov     eax, 1   ; 1 = sys_exit
         int     0x80
```

Компиляция и компоновка

```
$ nasm -f elf -o foo.o foo.asm
$ gcc -m32 -o foo foo.o         # компилируем с помощью gcc
```

Доступ к элементам данных

Доступ из ассемблера в C

Чтобы получить содержимое переменных C или объявить переменные, к которым C может обращаться, достаточно объявить имена как **EXTERN** или **GLOBAL**, соответсвенно .

Наличие начальных подчеркиваний зависит от соглашений C-компилятора. В частности, 16-битные компиляторы обычно требуют этого.

Компиляторы, генерирующие формат ELF обходятся без подчеркиваний.

Таким образом, переменная C, объявленная как **int var**, может быть доступна из ассемблера для 16-разрядных компиляторов как

```
extern  _var
        ...
        mov eax,[_var]
```

или

```
extern var
       ...
       mov eax,[var]
```

для компиляторов, генерирующих формат ELF.

Доступ из C в ассемблер

Чтобы объявить в ассемблерном коде целочисленную переменную, к которой программы на C могут обращаться как **`extern int var`**, обычно ее следует расположить в сегменте данных.

При использовании 16-разрядных компиляторов ее следует объявить с подчеркиваниями

```
          global _var
...
section  .data
_var      dd 0
```

а для генерирующих формат ELF без подчеркивания

```
          global var
...
section  .data
var       dd 0
```

Доступ из ассемблера к массиву в C

Чтобы получить доступ к массиву C, необходимо знать размер компонентов массива.

Например, переменные **int** имеют длину четыре байта (например), поэтому, если программа на C объявляет массив как **int a [10]**, доступ к элементу **[3]** можно получить, например, так

```
mov eax, [a+12] ; 12 = 3*4
```

Байтовое смещение, равное 12, получается умножением требуемого индекса массива, равного **3**, на размер элемента массива, равного 4, аналогично тому, как нужно делать в C, если мы желаем получить элемент не с помощью оператора **[]**, а через указатель .

Размеры базовых типов C в 32-разрядных компиляторах:

1 — для **char**;

2 — для **short int**;

4 — для **int**, **long int** и **float**;

8 — для **double**.

Указатели, являющиеся 32-битными адресами, также имеют длину 4 байта.

Некоторые компиляторы могут не следовать данным правилам, поэтому в программах на C следует использовать типы **stdint.h**. Это убережет от промахов при доступе к данным.

Представление данных

Segmetnt word size	32 bit						64 bit			
compiler	MS	Intel Win	Borla	Watc	GCC,Clang	Int Linux	MS	Intel Win	GCC,Clang	Int Linux
bool	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1
wchar_t	2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2
int	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	8	8
int64_t	8	8			8	8	8	8	8	8
enum (typical)	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8
long double	8	16	10	8	12	12	8	16	16	16
__m64	8	8			8	8		8	8	8
__m128	16	16			16	16	16	16	16	16
__m256	32	32			32	32	32	32	32	32
__m512	64	64			64	64	64	64	64	64
pointer	4	4	4	4	4	4	8	8	8	8
function pointer	4	4	4	4	4	4	8	8	8	8
data member ptr (min)	4	4	8	4	4	4	4	4	8	8
data member ptr (max)	12	12	8	12	4	4	12	12	8	8
member func ptr (min)	4	4	12	4	8	8	8	8	16	16
member func ptr (max)	16	16	12	16	8	8	24	24	16	16

Целые типы в C

Стандарт ISO/IEC 9899:2011: 7.20 Integer types <stdint.h>

Заголовочный файл **<stdint.h>** декларирует набор целых знаковых и беззнаковых типов фиксированной длины **[u]int*N*_t**, определяя соответствующие макросы.

```
int8_t
int16_t
int32_t
int64_t

uint8_t
uint16_t
uint32_t
uint64_t

...
```

Доступ из ассемблера к структуре данных в C

Чтобы получить доступ к структуре данных C, необходимо знать смещение от базы структуры до интересующего поля.

Это можно сделать, преобразовав определение структуры из C в определение структуры NASM (с помощью **STRUC**), или, прямо вычислить смещения и использовать их.

В любом случае, чтобы сделать либо первое либо второе, необходимо прочитать руководство C-компилятора, чтобы узнать, как он организует структуры данных.

NASM не предоставляет возможности особого выравнивания для элементов структуры в своем собственном макросе **STRUC**, поэтому выравнивание необходимо указывать отдельно, если C-компилятор C его генерирует.

Как правило, можно обнаружить, что структура, подобная

```
struct {
    char c;
    int  i;
} foo;
```

может быть длиной восемь байтов, а не пять, поскольку поле **int** будет выровнено по четырех-байтовой границе.

Однако такого рода особенности обычно настраиваются в C-компиляторе с использованием параметров командной строки или директив **#pragma**, поэтому иногда достаточно всего лишь выяснить, как это делает используемый компилятор.

c32.mac: вспомогательные макросы для 32-битного интерфейса C

В составе **NASM**, в каталоге **misc**, находится файл макросов **c32.mac**.

Он определяет три макроса: **proc**, **arg** и **endproc**. Они предназначены для использования в определениях процедур в стиле C, и позволяют автоматизировать большую часть работы, связанной с отслеживанием соглашений о вызовах.

Пример ассемблерной функции, использующей набор данных макросов:

```
proc    foo32

%$i     arg
%$j     arg
        mov     eax, [ebp + %$i]
        mov     ebx, [ebp + %$j]
        add     eax, [ebx]

endproc
```

Здесь символ **foo32** определен как процедура, принимающая два аргумента: первый (**i**) — целое число, второй (**j**) — указатель на целое число.

Процедура возвращает значение **i + *j**.

```
%imacro proc 1         ; начинает определение процедуры
%push proc
          global %1
%1:       push ebp
          mov  ebp,esp
%assign %$arg 8
%define %$procname %1 ; локальная в контексте метка
%endmacro
```

```
%imacro arg 0-1 4      ; используется с именем аргумента в качестве метки
%00     equ %$arg
%assign %$arg %1+%$arg
%endmacro
```

Следует обратить внимание, что макрос **arg** имеет **EQU** в качестве первой строки своего раскрытия, и поскольку метка перед вызовом макроса добавляется к первой строке расширенного макроса, **EQU** работает, определяя **%$i** как смещение от **EBP**.

```
%imacro endproc 0
%ifnctx proc                          ; в контексте ли proc?
%error Mismatched `endproc'/`proc'    ; нет, ошибка
%else                                 ; да, продолжаем
     leave
     ret
__end_%$procname:                     ; полезно для вычисления размера функции
%pop
%endif
%endmacro
```

Используется локальная переменная, сохраняемая макросом **proc** и восстанавливаемая макросом **endproc**, так что это же имя аргумента может использоваться в последующих процедурах.

Аргумент может принимать необязательный параметр, определяющий размер аргумента. Если размер не указан, предполагается 4, поскольку вполне вероятно, что многие параметры функции будут иметь тип **int** или тип указателя.

Сборка ассемблерного кода с libc

```
$ cat printf.asm
global   main   ; метка _start используется внутри обертки которую использует gcc
extern  printf  ; та самая printf(const char *format, ...)
;-----------------------------------------------------------
section .data
number  dd  12345
message db "int value: %d (12345)", 10, 0
;-----------------------------------------------------------
section  .text
_start:
main:
        push    dword [number]
        push    dword message
        call    printf
        add     esp, byte 8
;exit(0) -----------------------------------------------------
        mov     ebx, 0    ; NOERR exit code
        mov     eax, 1    ; 1 = sys_exit
        int     0x80
        xor     ebx, ebx ; exit code
        xor     eax, eax
        inc     eax       ; 1 = sys_exit
        int     0x80
```

Вызов ассемблера без явного указания имени выходного файла

```
$ nasm -f elf32 printf.asm ; -->  printf.o
```

Если нужен листинг

```
$ nasm -f elf32 -l printf.lst printf.asm
1                                  ;global   _start
2                                  global main
3                                  extern  printf
4                                  ;--------------------------------------
5                                  section .data
6 00000000 39300000                number  dd  12345
7 00000004 696E742076616C7565-     message db "int value: %d (12345)", 10, 0
7 0000000D 3A20256420283132333-
7 00000016 3435290A00
8                                  ;--------------------------------------
9                                  section   .text
10                                 _start:
11                                 main:
12 00000000 FF35[00000000]                 push    dword [number]
13 00000006 68[04000000]                   push    dword message
14 0000000B E8(00000000)                   call    printf
15 00000010 83C408                         add     esp, byte 8
16                                 ;exit(0) -----------------------------
17 00000013 BB00000000                     mov     ebx, 0   ; exit code
18 00000018 B801000000                     mov     eax, 1   ; 1 = sys_exit
19 0000001D CD80                           int     0x80

17 0000001F 31DB                           xor     ebx, ebx ; exit code
18 00000021 31C0                           xor     eax, eax
19 00000023 40                             inc     eax      ; 1 = sys_exit
19 0000001D CD80                           int     0x80
```

Компоновка

Используем gcc, который по умолчанию «прицепит» к нашему коду код для инициализации libc.

```
$ gcc -m32 printf.o -o printf
```

Запуск

```
$ ./printf
int value: 12345 (12345)
```

Попытка компоновки, используя «голый» `ld`

```
$ ld -m elf_i386 -e main printf.o -o printf -lc
$ ./printf
bash: ./printf: Нет такого файла или каталога
```

Отсутствует код времени исполнения, предназначенный для инициализации **`libc`**.

Системные вызовы и язык ассемблера

Номера системных вызовов определены в следующих файлах ядра:

```
/usr/include/asm-generic/unistd.h
/usr/include/asm/unistd_64.h
/usr/include/asm/unistd_x32.h

/usr/include/sys/syscall.h
```

Имена системных вызовов

`__NR_syscalls` — в ядре

`SYS_syscalls` — в glibc

Вызов в ядро из C

```
extern long int syscall(long int __sysno, ...);
```

Порядок загрузки регистров

EAX — номер системного вызова (**`long int __sysno`**)

EBX, ECX, EDX, ESI, EDI — параметры (**`...`**)

Преобразование заголовочных файлов C в заголовочные файлы nasm

Ряд заголовочных файлов для nasm получаются из заголовочных файлов Си методом фильтрации с помощью утилиты **sed** (stream editor). Утилита управляется скриптами, принимающими стандартные регулярные выражения. В частности, этим способом можно получить определения констант из **errno.h** и **unistd_32.h**.

Файл unistd_32.h

Содержит перечень номеров системных вызовов для использования в C.

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
...
#define __NR_rseq 386
...
#define __NR_pidfd_open 434
#define __NR_clone3 435
#define __NR_close_range 436
#define __NR_openat2 437
#define __NR_pidfd_getfd 438
#define __NR_faccessat2 439
#define __NR_process_madvise 440

#endif /* _ASM_X86_UNISTD_32_H */
```

Файл syscalls_32.inc

Содержит перечень номеров системных вызовов для использования в nasm.

```
%ifndef __NASM_SYSCALLS_32_INC__
__NASM_SYSCALLS_32_INC__    EQU 1
; Порядок загрузки параметров вызова:
; eax -- номер системного вызова
; ebx, ecx, edx, esi, edi -- первый, второй, ..., пятый параметры
SYS_restart_syscall EQU 0
SYS_exit            EQU 1   ; exit(int status)
SYS_fork            EQU 2
SYS_read            EQU 3   ; ssize_t read(int fd, void *buf, size_t count)
SYS_write           EQU 4   ; ssize_t write(int fd, const void *buf, size_t count)
SYS_open            EQU 5   ; int open(const char *pathname, int flags, [mode_t mode])
SYS_close           EQU 6   ; int close(int fd);
...
SYS_rseq            EQU 386 ; at 2019 last syscall
...
SYS_clone3          EQU 435
SYS_close_range     EQU 436
SYS_openat2         EQU 437
SYS_pidfd_getfd     EQU 438
SYS_faccessat2      EQU 439
SYS_process_madvise EQU 440

%endif ; __NASM_SYSCALLS_32_INC__
```

Скрипт для преобразования `.h` файлов в `.inc` файлы

Конвертирует заголовочный файл с командами C-препроцессора **`#define`** в определение констант nasm **`EQU и`** преобразует строки вида

```
#define  ENOENT     2 /* No such file or directory */
```

в строки вида

```
ENOENT       EQU  2 ; No such file or directory
```

sed-скрипт:

```
s/#define[[:space:]]*\([_[:alnum:]]*\)[[:space:]]*/\1\t\tEQU /
s/#/%/
s/\/\*/;/
s/\ \*\//
```

Для преобразования префикса **`__NR`** в **`SYS`** можно использовать sed-команду

```
s/__NR/SYS/
```

Красным цветом отмечены чисто синтаксические детали sed-команд.
Символ «/» может быть заменен другим, например, «|» или «+»

Преобразование errno.h в errno.inc

#include <errno.h> в конце концов подключает пару файлов

```
/usr/include/asm-generic/errno-base.h
/usr/include/asm-generic/errno.h
```

errno-base.h содержит основные коды ошибок от 1 до 34.

errno.h содержит коды ошибок от 35 до 133+.

```
$ cat errno-base.h
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H
#define  EPERM      1 /* Operation not permitted */
#define  ENOENT     2 /* No such file or directory */
....
#define  ERANGE    34 /* Math result not representable */
#endif

$ cat errno-base.inc
; SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
%ifndef _ASM_GENERIC_ERRNO_BASE_H
%define _ASM_GENERIC_ERRNO_BASE_H

EPERM      EQU  1  ; Operation not permitted
ENOENT     EQU  2  ; No such file or directory
...
ERANGE     EQU 34  ; Math result not representable

%endif
```

Соответствие директив C-препроцессора директивам препроцессора NASM

C	NASM
#if	%if
#ifdef	%ifdef
#ifndef	%ifndef
#elif	%elif
#else	%else
#endif	%endif
#include	%include
#define	%define, %xdefine
#undef	%undef
#line	
#error	%error
#pragma	

Соответствие C-макросов макросам препроцессора NASM

C	Nasm
__DATE__	__DATE__
__TIME__	__TIME__
__FILE__	__FILE__
__LINE__	__LINE__
__STDC__	
__STDC_HOSTED__	
__STDC_VERSION__	
__STDC_*	

Потоковый и файловый ввод-вывод

Процесс, загруженный для выполнения, изначально имеет доступ к трем открытым файловым дескрипторам, которые связаны со стандартными потоками ввода-вывода:

0 — `stdin`

1 — `stdout`

2 — `stderr`

Используя системные вызовы:

```
#define __NR_read   3
#define __NR_write  4
```

мы можем в программе на ассемблере читать из **`stdin`** и писать в **`stdout`** и **`stderr`**. При необходимости взаимодействовать с файлами и каталогами, а также прочими файловыми дескрипторами мы можем напрямую использовать системные вызовы

```
#define __NR_open    5 // open(2)
#define __NR_close   6 // close(2)
#define __NR_creat   8 // creat(2)
#define __NR_lseek  19 // lseek(2)
```

Системные вызовы из C/C++

Чтобы иметь возможность вызывать системные функции ядра из программ, написанных на C и C++ следует подключить файл

/usr/include/sys/syscall.h

```
#ifndef _SYSCALL_H
#define _SYSCALL_H 1

/* Этот файл должен содержать список номеров системных вызовов, которые знает
система. Но вместо дублирования мы используем информацию, доступную из исходных кодов
ядра. */
#include <asm/unistd.h>
#ifndef _LIBC
/* Файл заголовков ядра Linux определяет макросы `__NR_<name>', но некоторые
программы ожидают традиционную форму `SYS_<name>'.
Поэтому, при сборке libc мы сканируем список вызовов ядра и создаем файл
<bits/syscall.h> с макросами для всех имен `SYS_'. */
# include <bits/syscall.h>
#endif
#endif /* _SYSCALL_H */
```

В зависимости от разрядности генерируемого кода **asm/unistd.h** подключает либо **asm/unistd_32.h**, либо **asm/unistd_64.h**.

```
/* SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note */
#ifndef _ASM_X86_UNISTD_H
#define _ASM_X86_UNISTD_H

/* x32 syscall flag bit */
#define __X32_SYSCALL_BIT 0x40000000

# ifdef __i386__
#  include <asm/unistd_32.h>
# elif defined(__ILP32__)     /* IBM ILP32 compiler
#  include <asm/unistd_x32.h>
# else
#  include <asm/unistd_64.h>
# endif

#endif /* _ASM_X86_UNISTD_H */
```

```
$ cat /usr/include/asm/unistd_32.h | grep '__NR_' | wc
    439    1317   11822
$ cat /usr/include/asm/unistd_64.h | grep '__NR_' | wc
    361    1083    9633
$ rpm -qf /usr/include/asm/unistd_32.h
kernel-headers-5.16.5-100.fc34.x86_64
```

extern long int syscall(long int __sysno, ...)

Файл **unistd_32.h**, входящий в состав пакета kernel-headers-5.16.5-100.fc34.x86_64, содержит 439 определений имен **__NR_<syscall>** системных функций

```
$ cat /usr/include/asm/unistd_32.h | grep __NR | wc
    439    1317   11822
```

Не все из них реально реализованы в ядре.

Ниже для справки приведены нереализованные системные вызовы.

Они всегда возвращают **-1** и устанавливают для **errno** значение **ENOSYS**.

```
afs_syscall fdetach     getpmsg     lock        prof        putpmsg     tuxcall
break       ftime       gtty        madvise1    profil      security    ulimit
fattach     getmsg      isastream   mpx         putmsg      stty        vserver
```

Системные вызовы можно инициировать из программы на C/C++, используя непрямой системный вызов **syscall()**:

```
#include <unistd.h>
#include <sys/syscall.h>   /* для определений SYS_xxx */

extern long int syscall(long int __sysno, ...);
```

__sysno — код системного вызова.

`...` — переменное число параметров, которые следует передать системному вызову

```
$ cat unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
...
#define __NR_fsmount 432
#define __NR_fspick 433
#define __NR_pidfd_open 434
#define __NR_clone3 435
#define __NR_openat2 437
#define __NR_pidfd_getfd 438
#define __NR_faccessat2 439
#define __NR_process_madvise 440

#endif /* _ASM_X86_UNISTD_32_H */
```

Макросы. Файл syscalls.mac

```
;-----------------------------------------------------------------
; SYS_EXIT -- завершение вызывающего процесса
;-----------------------------------------------------------------
; void exit(int status)
;-----------------------------------------------------------------
; SYS_EXIT status
;       status -- код  завершаемого процесса
;-----------------------------------------------------------------

%macro  SYS_EXIT 1.nolist
        mov     ebx, %1         ; status
        mov     eax, SYS_exit
        int     0x80
%endmacro
```

```
;-----------------------------------------------------------------
; _syscall_write -- вывод буфера в файл
;-----------------------------------------------------------------
; ssize_t write(int fd, const void *buf, size_t count)
;-----------------------------------------------------------------
; _syscall_write fd, buf, count
;       fd    -- дескриптор открытого файла
;       buf   -- адрес буфера
;       count -- количество байт, подлежащих выводу
;-----------------------------------------------------------------
; Важно!!! Не сохраняет регистры
;-----------------------------------------------------------------
%macro  _syscall_write 3.nolist
        mov     eax, SYS_write
        mov     ebx, %1         ; fd
        mov     ecx, %2         ; buf
        mov     edx, %3         ; count
        int     0x80
%endmacro
```

```
; PUTS -- вывод строки на stdout
;-----------------------------------------------------------
; 1) PUTS buffer, buffer_sz
;      buffer    -- адрес (метка)
;      buffer_sz -- размер (число)
; 2) PUTS "строка символов", ...
; 3) PUTS число, ...
;      число -- трактуется, как один ASCII-символ
;-----------------------------------------------------------
%macro  PUTS    1+.nolist
                push    ebx
                push    ecx
                push    edx
;-----------------------------------------------------------
; 1) параметр -- идентификатор
;-----------------------------------------------------------
%ifid   %1
                _syscall_write  1, %1
```

```
;-----------------------------------------------------------------
; 2) параметр – "строка символов"
;-----------------------------------------------------------------
%elifstr  %1
section         .data
%%string         db      %1, 0
%%strlen         equ     $ - %%string
section         .text
                 _syscall_write  1, %%string, %%strlen
```

```
;-----------------------------------------------------------------
; 3) параметр -- число
;-----------------------------------------------------------------
%elifnum  %1
section         .data
%%c              db      %1%256
section         .text
                 _syscall_write  1, %%c, 1
%endif
                 pop     edx
                 pop     ecx
                 pop     ebx
%endmacro
```

Вывод строки на stdout

```
%include        "syscalls_32.inc"
%include        "syscalls.mac"

global  _start
;-----------------------------------------------------------------
section         .data
;-----------------------------------------------------------------
message         db "Язык ассемблера", 10, 0
message_sz      equ $ - message
;-----------------------------------------------------------------
section         .text
;-----------------------------------------------------------------
_start:
                PUTS    "NASM version 2.14.02 compiled on Mar 19 2020"
                PUTS    message, message_sz
                SYS_EXIT 0
```

Компиляция, компоновка и запуск исполняемого модуля

```
$ nasm -f elf -i../include/ -o puts.o puts.asm
$ ld -m elf_i386 puts.o  -o puts
$ ./puts
NASM version 2.14.02 compiled on Mar 19 2020
Язык ассемблера
$
```