

Юрий Анатольевич Богоявленский, заведующий кафедрой
Информатики и математического обеспечения, к.т.н., доцент:
ybgv@cs.petrus.ru, <https://cs.petrus.ru>

Веб страница дисциплины

<https://cs.petrus.ru/~chistyak/architecture/>

Лекция 7. Стековый доступ к оперативной памяти. Функции и соглашения ABI

План лекции

Стековый доступ к оперативной памяти.....	1
Определение и иллюстрация работы стека.....	2
Характеристики архитектурного стека в IA-32.....	3
Команды работы со стеком.....	3
Адресный доступ к элементам стека.....	4
Пример работы стека и адресного доступа к его элементам.....	6
Функции и соглашения ABI.....	7
Общие положения.....	7
Соглашения о вызовах функций. Стандарт ABI.....	8
Основные элементы функции и ABI.....	11
Порядок передачи параметров функции в стек.....	12
Выводы.....	13

Стековый доступ к оперативной памяти

Стек (от англ. **stack** — стопка) это пожалуй наиболее популярная структура данных, применяемая при решении самых разнообразных задач. В далеко неполный их список входят разного рода рекурсивные вычисления, синтаксический анализ языков программирования, поиск с возвратом, управление памятью, соглашения о вызовах функций. Существуют стековые языки программирования (например **Forth**). Такая популярность объясняется простотой доступа к элементам стека и малым временем вычисления адреса его элементов в ОП.

NB. Мы будем рассматривать применение стека при реализации стандартных соглашений о вызовах функций.

Дисциплина доступа к элементам стека обозначается аббревиатурой **LIFO** (**Last In First Out** — последний записанный элемент структуры читается первым). Конструкции, идейно аналогичные стеку были известны в технике (например магазины в многозарядном огнестрельном оружии). Стек также применяется в теории алгоритмов и автоматов, где он часто называется магазином (например, в конечных автоматах с магазинной памятью).

Определение и иллюстрация работы стека

Определение. Стек это совокупность линейно связанных однородных элементов в которой операция записи нового элемент всегда происходит в начало стека (т. н. вершину), а операция чтения получает элемент также из начала стека. Стандартно операция записи в стек называется **push** (поместить), а операция чтения — **pop** (извлечь).

Из этого определения вытекает, что операции **push** и **pop** имеют всего один операнд: для **push** — значение, помещаемое в стек, для **pop** — указатель места, получающего значение из вершины стека.

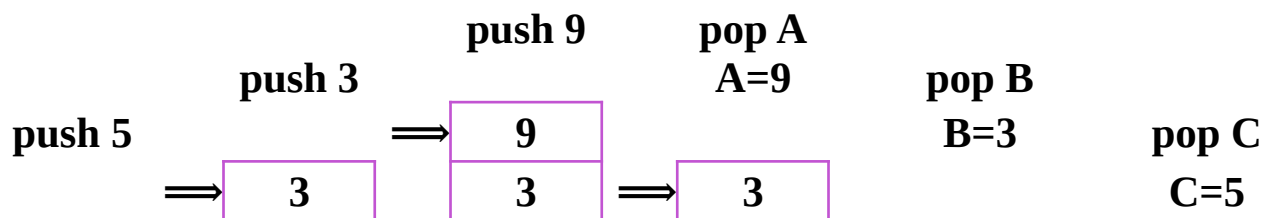




Рис. 1. Иллюстрация работы стека.

Слева показано исходное состояние — в стеке находится единственное значение — 1. Символ $\Rightarrow$ указывает на положение вершины при последовательном слева направо выполнении операций, указанных над таблицами, содержащими значения элементов в стеке.

Характеристики архитектурного стека в IA-32

Стек настолько хорошо себя зарекомендовал, что в архитектурах практически всех процессоров он, для повышения быстродействия, реализован аппаратно и называется — архитектурный стек.

В лекциях и лабораторных работах мы будем считать, что элементами стека могут быть ТОЛЬКО четырех байтовые двойные слова и только они могут быть операндами операций записи и чтения (вообще говоря это не всегда так). При этом:

- Стеком управляет регистр `%esp` (`extended stack pointer`) в котором содержится четырех байтовый адрес вершины стека в ОП.
- При помещении значения в стек значение `%esp` уменьшается на четыре, говорят, что «стек растет в направлении меньших адресов».
- При извлечении значения из стека значение `%esp` увеличивается на четыре.
- При запуске программы системный вызов `execve`:
 - выделяет в конце доступный программе ОП блок для стека;
 - присваивает регистру `%esp` значение, указывающее на двойное слово этого блока с наибольшим адресом — вершину стека;
 - присваивает этому двойному слову значение 1.

Команды работы со стеком

- `pushl <операнд R/M>` — поместить `<операнд R/M>` в вершину стека.

Действие команды эквивалентно действию пары команд:

```
subl    $4,%esp
```

```
movl    <операнд R/M>,(%esp)
```

- `popl <операнд R/M>` — извлечь значение из вершины стека и записать его в <операнд R/M>.

Действие команды эквивалентно действию пары команд:

```
movl    (%esp),<операнд R/M>
```

```
addl    $4,%esp
```

- `pusha` — поместить в стек значения восьми регистров общего назначения (РОН) в порядке `%eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, %edi`.

NB. В стек записывается значение `%esp`, которое он имел ДО выполнения команды `pusha`. В остальном действие команды эквивалентно выполнению восьми команд `pushl` для каждого регистра.

- `pora` — извлечь из стека и поместить в РОН, в порядке, обратном принятому в команде `pusha` значения восьми последовательных элементов стека, начиная с его вершины.

NB. Важное преимущество команд `pushl` и `popl` — отсутствие необходимости указывать в команде адрес записи/чтения значения, находящегося на вершине стека.

Адресный доступ к элементам стека

Команда `popl` извлекает значение из вершины стека, что делает недоступным его в ОП. Это означает, что если необходимо многократное обращение к значениям в стеке с сохранением его текущей структуры, т.е. без изменения этих значений и регистра `%esp` (например для элементов кадра стека (рассмотрим позже), формируемых при вызове функции), то для доступа к ним следует использовать другие механизмы.

Оказывается, что такой многократный доступ можно легко и эффективно организовать с помощью механизма режимов адресации данных, рассмотренного ранее. Покажем как это делается.

Прежде всего отметим, что в каждый момент времени стек можно рассматривать как одномерный массив и, следовательно, получать адреса его элементов в ОП с помощью регистровой адресации. Естественно при этом использовать в качестве базового регистр `%esp`.

Предположим, что показанные на Рис. 1 состояния стека являются состояниями архитектурного стека, полученными командами `pushl` и `popl`. На рисунке ниже дано детализированное представление состояния стека после выполнения команды `pushl $9`.

Элемент на вершине стека		Адреса байтов элемента относительно значения в %esp				Форма операнда элемента стека в режиме регистровой адресации	Значение ЕА по формуле регистровой адресации
⇒%esp	9	0	1	2	3	0(%esp)	0 + 3n(%esp)
		9					
	3	4	5	6	7	4(%esp)	4 + 3n(%esp)
		3					
5	8	9	10	11	8(%esp)	8 + 3n(%esp)	
		5					
1	12	13	14	15	12(%esp)	12 + 3n(%esp)	
		1					

Рис. 2. Адресный доступ к элементам стека.

В левом столбце приведено рассматриваемое состояние стека. В следующем столбце для каждого байта каждого элемента стека даны величины на которые нужно увеличит значение `%esp`, чтобы получить адрес ОП этого байта. В следующем столбце дана форма операнда регистровой адресации, обеспечивающая многократный адресный доступ к соответствующему элементу стека. В последнем столбце приводится формула вычисления эффективного адреса этого элемента.

Пример работы стека и адресного доступа к его элементам.

```
.include "my-macro"    # подключение файла с  
макроопределениями
```

```
.data
```

```
L:  .long  5  
L1: .long  0  
L2: .long  0  
L3: .long  0  
L4: .long 100
```

```
Ad1:  .long 0  
Ad2:  .long 0  
Ad3:  .long 0  
Ad4:  .long 0  
Ad5:  .long 0
```

```
.text
```

```
.global _start  
_start:  
    nop
```

```
#    Запись элементов в стек
```

```
    pushl $3      # непоср. операнд в стек  
    pushl $2  
    pushl L       # операнд из ОП в стек  
    pushl L4
```

```
#    Многократный адресный доступ к элементам в стеке
```

```
    movl 0(%esp),%eax # из вершины стека в ОП  
    movl %eax,Ad1     # из вершины стека в ОП
```

```
    movl 0(%esp),%eax # из вершины в другой адр. ОП  
    movl %eax,Ad2     # из вершины в другой адр. ОП
```

```
    movl 4(%esp),%eax # из элем ниже вершины в ОП  
    movl %eax,Ad3     # из элем ниже вершины в ОП
```

```

movl    8(%esp),%eax    # из след. ниже в ОП
movl    %eax,Ad4        # из след. ниже в ОП

movl    12(%esp),%eax   # из самого нижнего в ОП
movl    %eax,Ad5        # из самого нижнего в ОП

# Чтение элементов из стека

popl    L1              # из вершины стека в ОП
popl    L2
popl    L3
popl    %ecx            # из вершины стека в регистр

# Запись и чтение POH

movl    $0x10,%eax      # для демонстрации
pusha
movl    $0x20,28(%esp)  # изменили знач %eax в стеке
popa

Finish
.end

```

Функции и соглашения ABI

Общие положения

Ранее мы рассмотрели концепцию макропрограммирования для борьбы со сложностью программных систем. Еще одна популярная концепция для решения этой задачи — концепция функции. Здесь также можно задать имя фрагмента кода и способ его модернизация для использования этого имени (от одной до нескольких строк) для выполнения кода модернизированного фрагмента без повторного размещения программистом его текста в исходном коде.

При этом подстановка кода функции в исходный текст, как при микропрограммировании, не применяется. Запуск функции на выполнение называется **ВЫЗОВОМ ФУНКЦИИ** и реализуется с помощью специальных команд

процессора путем передача управления на код ее двоичных команд и возврата управления по завершению ее работы.

Функции представляют собой удобный механизм разбиения исходного текста на большое количество небольших по объёму модулей, которые можно легко контролировать, отлаживать и тестировать. Например известный программист Джерард Дж. Хольцманн (Gerard J. Holzmann) работая в NASA, в 2006 г. сформулировал нацеленные на уменьшения ошибок правила надежного программирования, четвертое из которых гласит:

Любая функция или метод после распечатки должны уместиться на стандартном листе бумаги (имеется в виду формат A4 — 210 x 297 мм). При этом для каждого оператора и каждого объявления переменной отводится отдельная строка. Таким образом, размер функции не превысит 50-60 операторов.

Содержательно каждая функция должна представлять собой логический модуль кода, который понимается и модифицируется как единый блок. Функция должны выполнять в некотором смысле «изолированный» алгоритм, который имеет небольшое количество входных и выходных данных. Гораздо труднее понять логический блок, который занимает несколько экранов или несколько страниц при печати. Чрезмерно длинные функции часто свидетельствуют о том, что программа плохо структурирована.

NBNB. Искусство разбивать программу на функции является базовой частью компетенции программиста. Примерами хорошо сконструированных систем функций являются системные вызовы и команды `shell`.

NBNB. Одно из важных преимуществ функций — возможность многократного повторного использования. Функции группируются в библиотеки. Одна из фундаментальных — библиотека `glibc`, позже кратко рассмотрим ее.

Поскольку модули могут транслироваться отдельно, необходим механизм передачи исходных данных в модуль вызываемой функции и передачи результата её работы в модуль вызывающей функции. При этом исходные данные называются параметрами функций.

Соглашения о вызовах функций. Стандарт ABI

Механизм:

- передачи параметров в функцию;
- передачи управления в тело функции;
- возврата управления вызывающей функции и передачи ей результатов работы

вызываемой функции.

называется соглашениями о вызовах функций (`calling conventions`).

Существует несколько различных стандартов этих соглашений, например `stdcall` для Win32 API или `vectorcall` для ускорения передачи векторных параметров в процессорах, имеющих набор SIMD команд SSE2 и выше.

Мы рассмотрим соглашения определенные в документе

`SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement Fourth Edition`, кратко называемом ABI (Двоичный интерфейс приложений), доступном по ссылке:

<https://kappa.cs.petrus.ru/~chistyak/architecture/docs/abi386-4.pdf>

(см. стр. 35).

Эти соглашения используются как в языке ассемблера так и в языке C (для него эти соглашения обозначаются аббревиатурой `cdecl`), что позволяет в одном исполняемом модуле использовать функции, написанные на обоих языках и, тем самым, повышать эффективность и производительность программ. Большинство современных реализаций других языков также поддерживают правила соглашения ABI о вызовах.

Отметим, что сохранение и восстановление значений РОН не входят в соглашения но рекомендуются т. к. позволяет свободно изменять значения РОН в вызываемой функции.

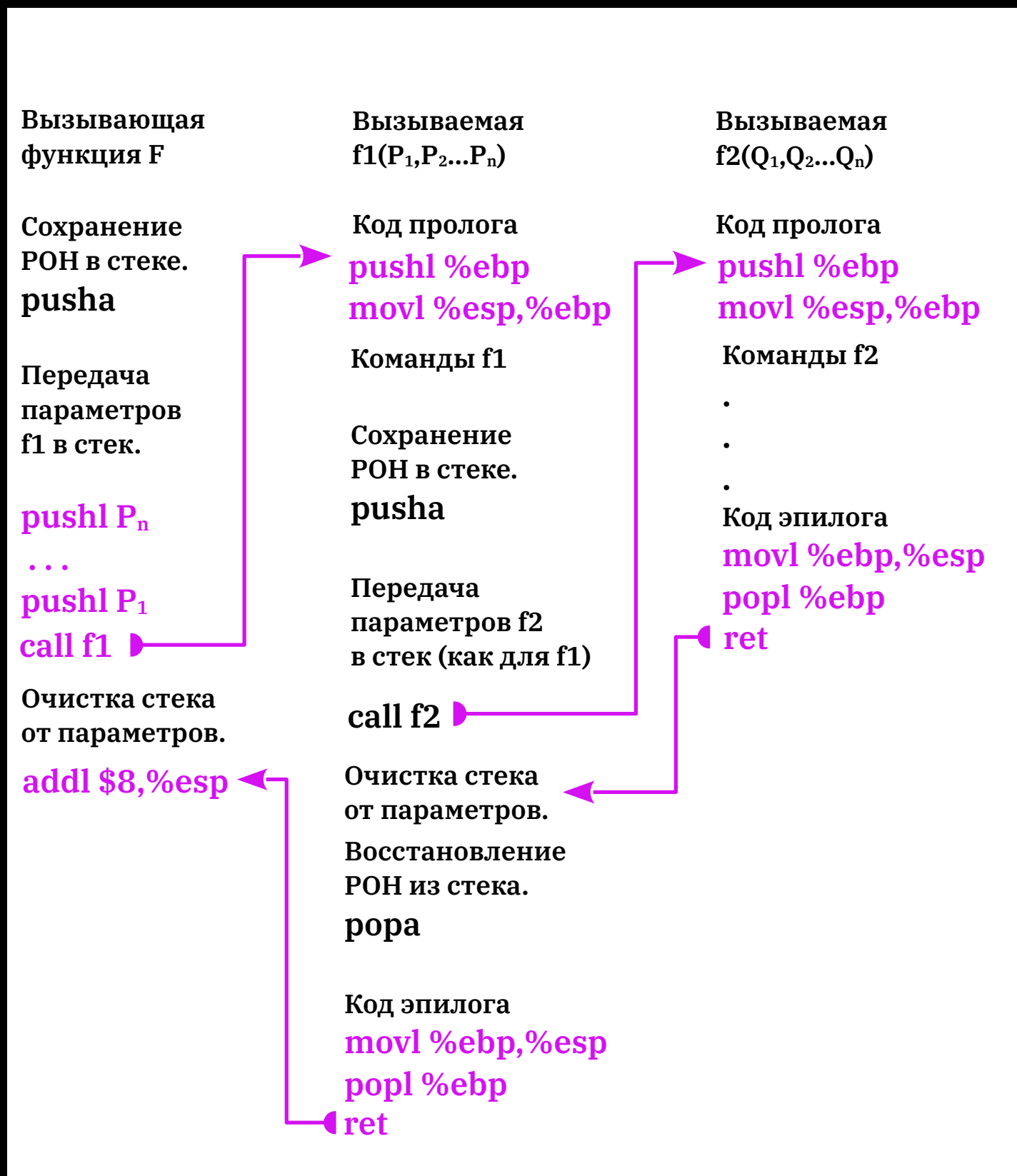


Рис. 3. Схема организации функций по соглашениям о вызовах ABI.

- Имя функции – символьное имя, которое при ассемблировании получает значение адреса точки входа в функцию.
- Вызывающая функция (`caller`) — функция, выполняющая команду `call`.
- Вызываемая функция (`callee`) — функция, которой передает управление команда `call`.
- Параметры функции – единицы входных или выходных данных, значения которых должна получать, обрабатывать и возвращать функция. Вызывающая функция должна поместить их значения в аппаратный стек.
- `call` — команда передачи управления в точку входа в функцию.
- Адрес возврата – адрес команды, следующей за командой `call`, которая при выполнении вычисляет его и записывает в аппаратный стек.
- `ret` — команда, которая должна быть выполнена последней в вызываемой функции. Она читает из стека адрес возврата и передает управление в вызывающую функцию на этот адрес.
- По соглашениям после завершения работы функции удалить ее параметры из стека должна вызывающая программа
- Код пролога — две стандартные команды, которые должны быть выполнены первыми при входе в функцию.
- Код эпилога — две стандартные команды, которые должны быть выполнены в функции непосредственно перед выполнением каждой командой `ret`.
- Коды пролога и эпилога используют регистры `%ebp` и `%esp`, обеспечивая вместе с командами `call` и `ret` произвольную глубину вложенности вызовов функций.

NBNB. По соглашениям **ABI** результат работы функции (**Return value**) в виде четырех байтового целого или адреса (например структуры или массива) передается в вызывающую программу путем записи этого значения в регистр `%eax` перед выполнением команды `ret`. Мы не рассматриваем здесь способы передачи в вызывающую программу результатов других типов.

Порядок передачи параметров функции в стек

Пусть запись функции с N параметрами на языке C имеет вид:

$f(P_1, P_2 \dots P_N)$. По соглашениям о вызовах, значения параметров нужно передавать стек в порядке ОБРАТНОМ представленному в этой записи в следующей последовательности:

`pushl PN`

`pushl PN-1`

`...`

`pushl P1`

Выводы

- Стек (от англ. *stack* — стопка) это пожалуй наиболее популярная структура данных, применяемая при решении самых разнообразных задач.
- Дисциплина доступа к элементам стека обозначается аббревиатурой LIFO (Last In First Out — последний записанный элемент структуры читается первым)..
- Стек это совокупность линейно связанных однородных элементов в которой операция записи нового элемент всегда происходит в начало стека (т. н. вершину), а операция чтения получает элемент также из начала стека.
- Стек настолько хорошо себя зарекомендовал, что в архитектурах практически всех процессоров он, для повышения быстродействия, реализован аппаратно и называется — архитектурный стек.
- Стеком управляет регистр `%esp` (extended stack pointer) в котором содержится четырех байтовый адрес вершины стека в ОП.
- При помещении значения в стек значение `%esp` уменьшается на четыре, говорят, что «стек растет в направлении меньших адресов». При извлечении значения из стека значение `%esp` увеличивается на четыре.

- Адресный доступ к элементам стека можно легко и эффективно организовать с помощью механизма режимов адресации данных.

- Функции представляют собой механизм разбиения исходного текста на большое количество небольших по объёму модулей, которые можно легко контролировать, отлаживать и тестировать.

- Правило Хольцмана. Любая функция или метод после распечатки должны уместиться на стандартном листе бумаги (имеется в виду формат А4 — 210 x 297 мм). При этом для каждого оператора и каждого объявления переменной отводится отдельная строка. Таким образом, размер функции не превысит 50-60 операторов.

- Искусство разбивать программу на функции является базовой частью компетенции программиста.

- Механизм:

- передачи параметров в функцию;
- передачи управления в тело функции;
- возврата управления вызывающей функции и передачи ей результатов работы вызываемой функции.

называется соглашениями о вызовах функций (`calling conventions`). Существует несколько различных стандартов этих соглашений.

- Мы рассматриваем соглашения определенные в документе `SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement Fourth Edition`, кратко называемом `ABI` (Двоичный интерфейс приложений).

- Эти соглашения используются как в языке ассемблера так и в языке `C` (для него эти соглашения обозначаются аббревиатурой `cdecl`), что позволяет в одном исполняемом модуле использовать функции, написанные на обоих языках.

- Элементы функции: имя функции, вызывающая функция (`caller`), вызываемая функция (`callee`), параметры функции, `call` — команда передачи управления в точку входа в функцию, адрес возврата, `ret` — команда, которая должна быть выполнена последней в вызываемой функции.

- Элементы **ABI**: передача параметров через стек, код пролога, код эпилога.

- **ABI** обеспечивает произвольную глубину вложенности вызовов функций,

ограниченную только объемом ОП, выделенным для стека.