

Лекция 11

19 марта

Языки программирования (ЯП), базирующиеся на стеке вызовов

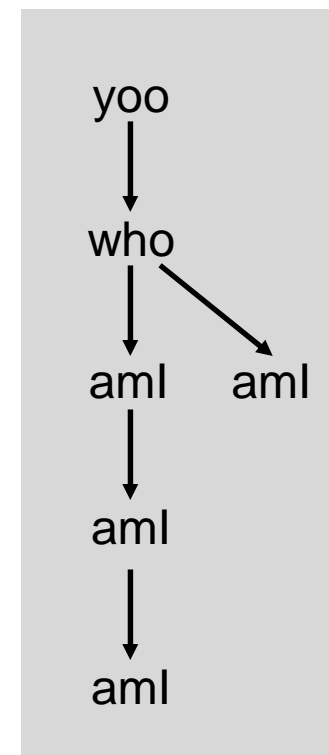
- ЯП с поддержкой рекурсии
 - C, Pascal, Java
 - Код функции можно вызывать повторно (“*Reentrant*”)
 - Одновременно могут выполняться несколько вызовов функции
 - Необходимо выделять память под сохранение состояния каждого работающего вызова
 - Аргументы
 - Локальные переменные
 - Адрес возврата
- Стек
 - Сохранять состояние вызова функции надо в ограниченный период времени: от момента вызова до момент выхода
 - Вызываемая функция всегда завершается до вызывающей
- Стек выделяется **Фреймами**
 - Состояние отдельного вызова функции

Пример цепочки вызовов

```
yoo (...)  
{  
  .  
  .  
  who ( ) ;  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ( ) ;  
  . . .  
  amI ( ) ;  
  . . .  
}
```

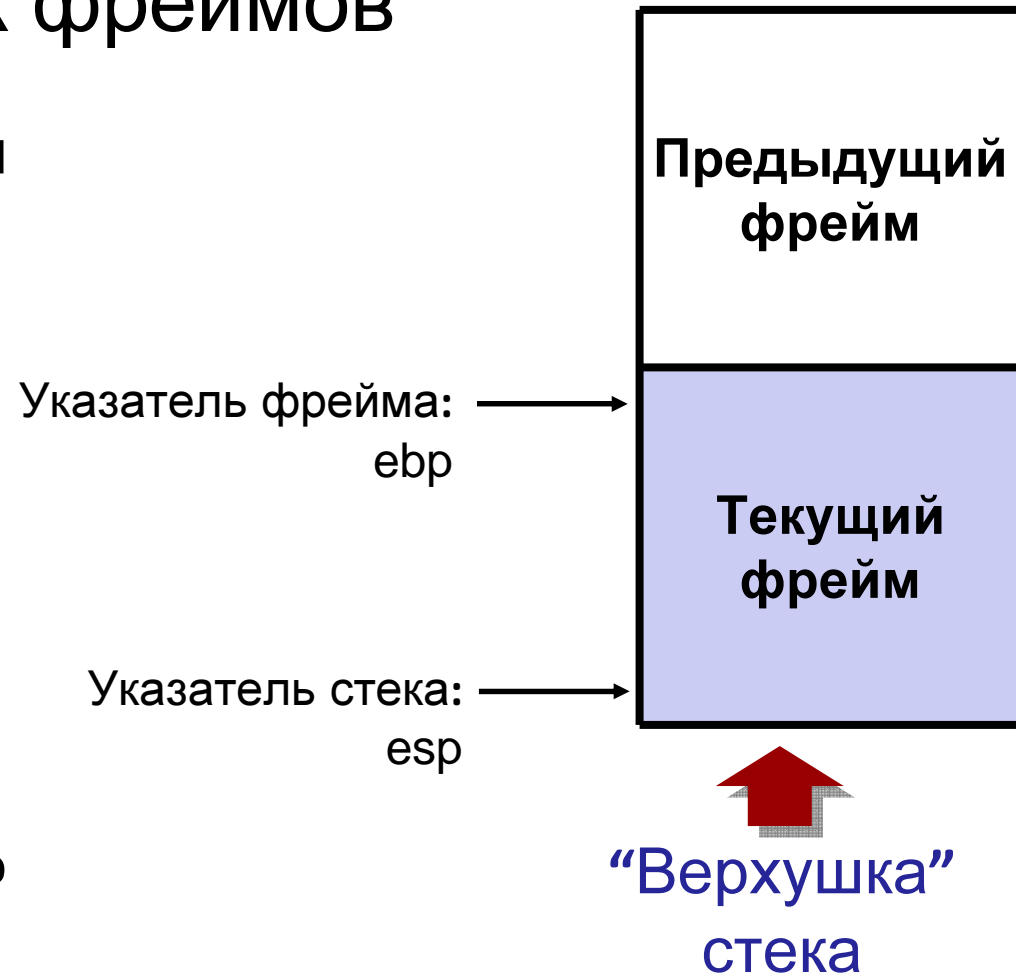
```
amI (...)  
{  
  .  
  .  
  amI ( ) ;  
  .  
  .  
}
```




Функция amI() рекурсивная

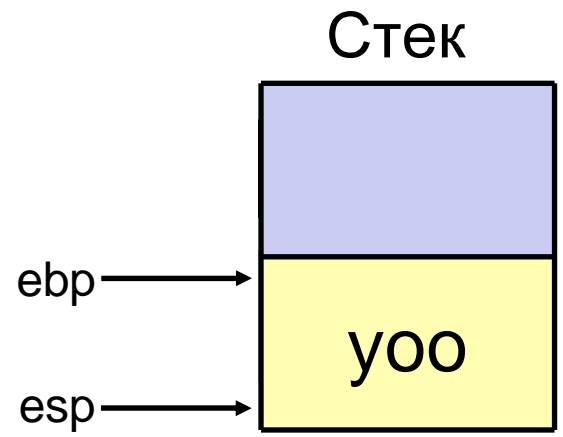
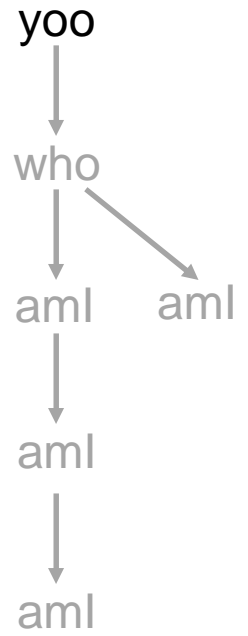
Стек фреймов

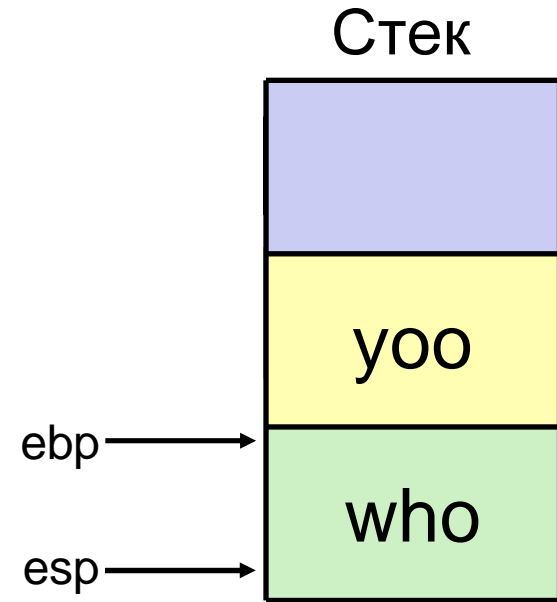
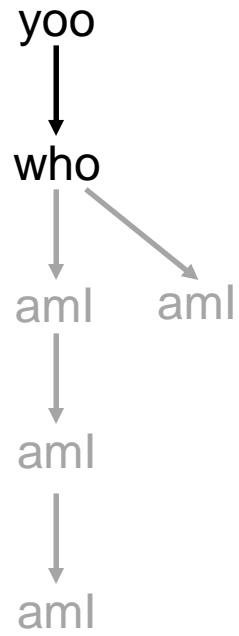
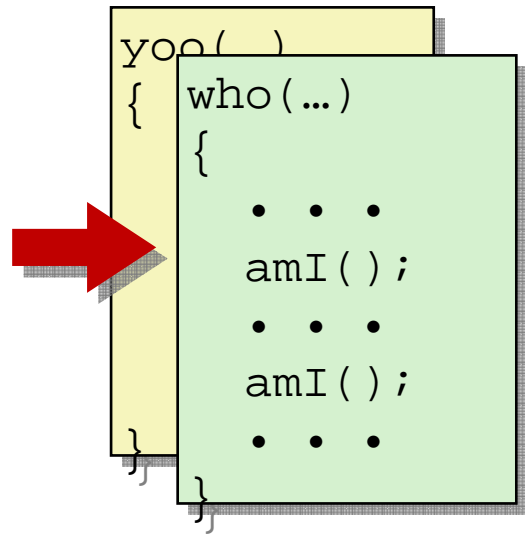
- Во фрейме размещаются
 - Локальные переменные
 - Данные, необходимые для возврата из функции
 - Временные переменные
- Управление фреймами
 - Пространство выделяется во время входа в функцию
 - «пролог» функции
 - Освобождается на выходе
 - «эпилог» функции

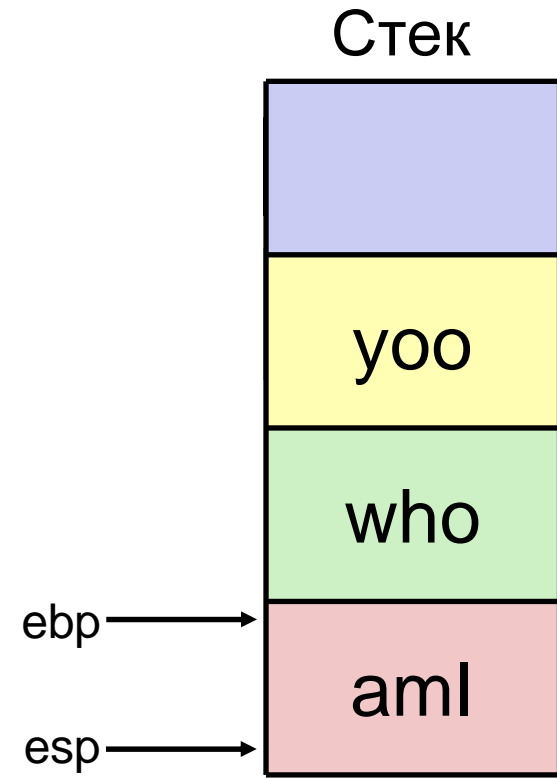
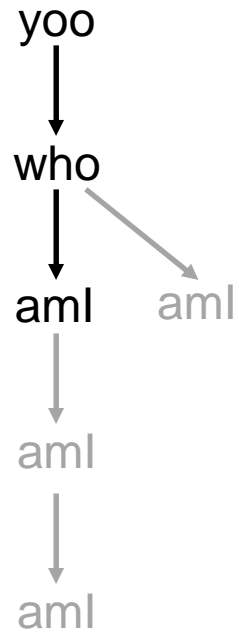
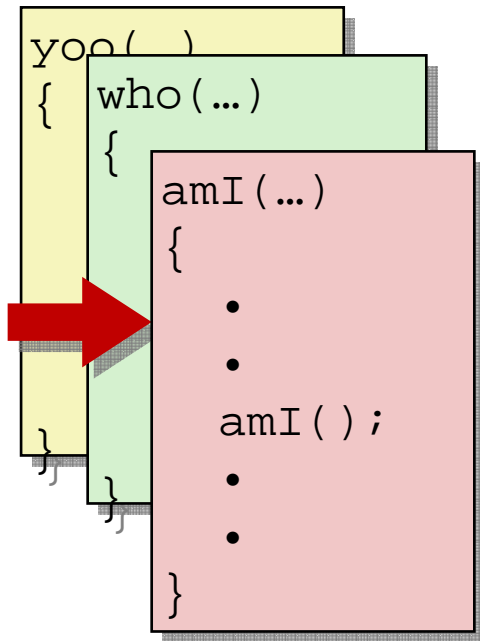


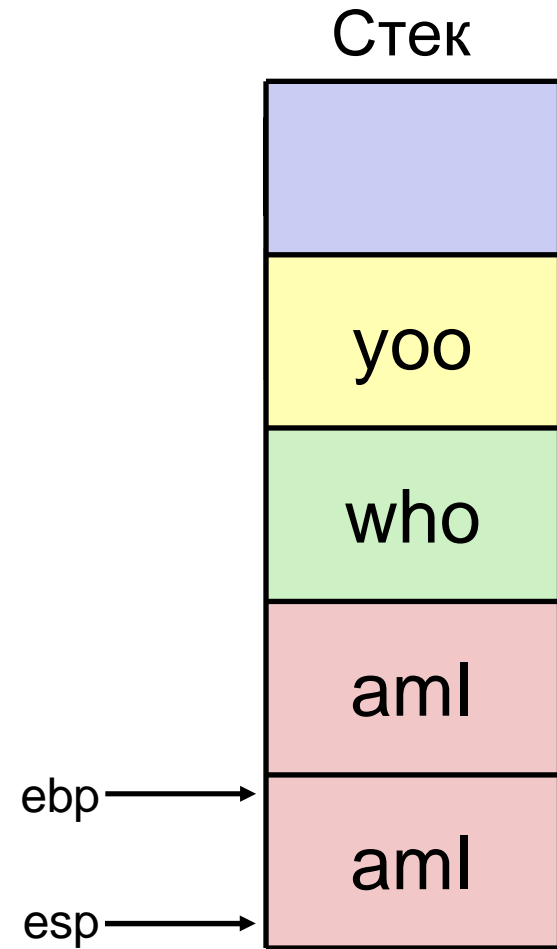
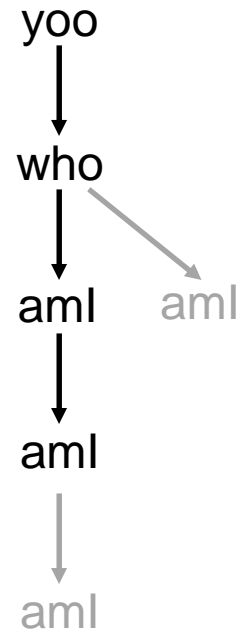
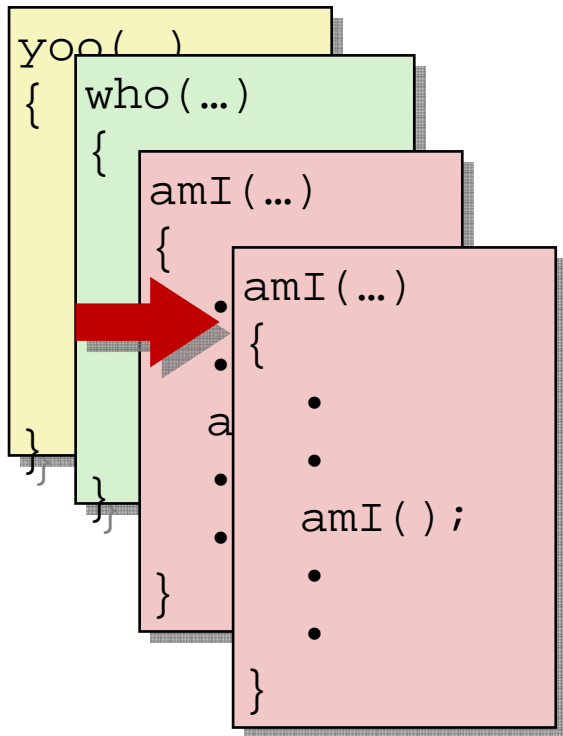


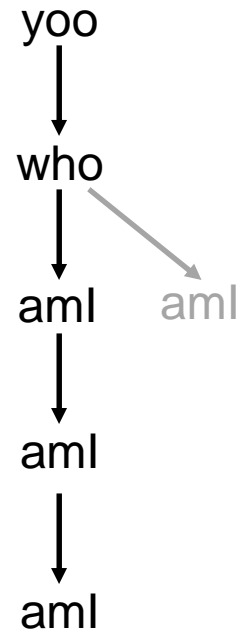
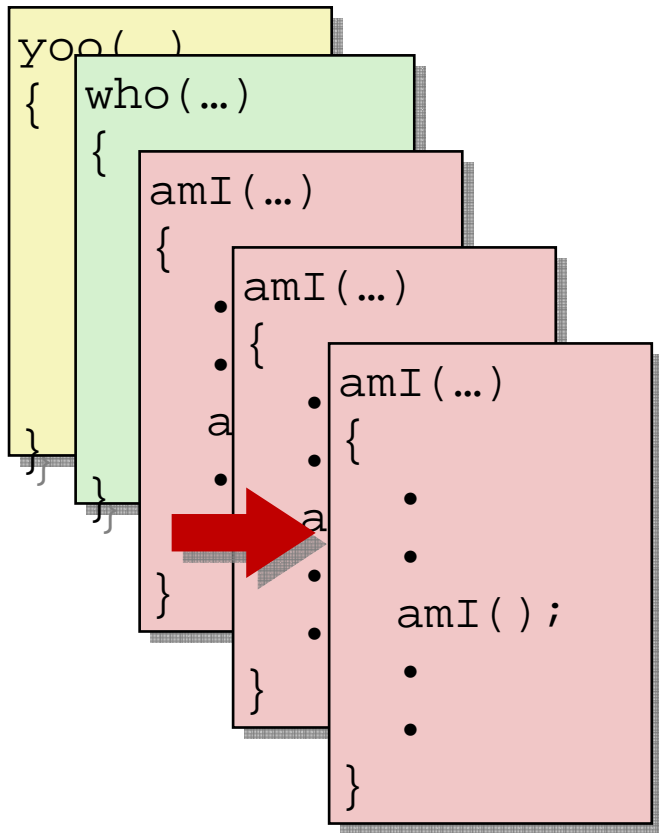
```
yoo(...)  
{  
  •  
  •  
  who( ) ;  
  •  
  •  
}
```

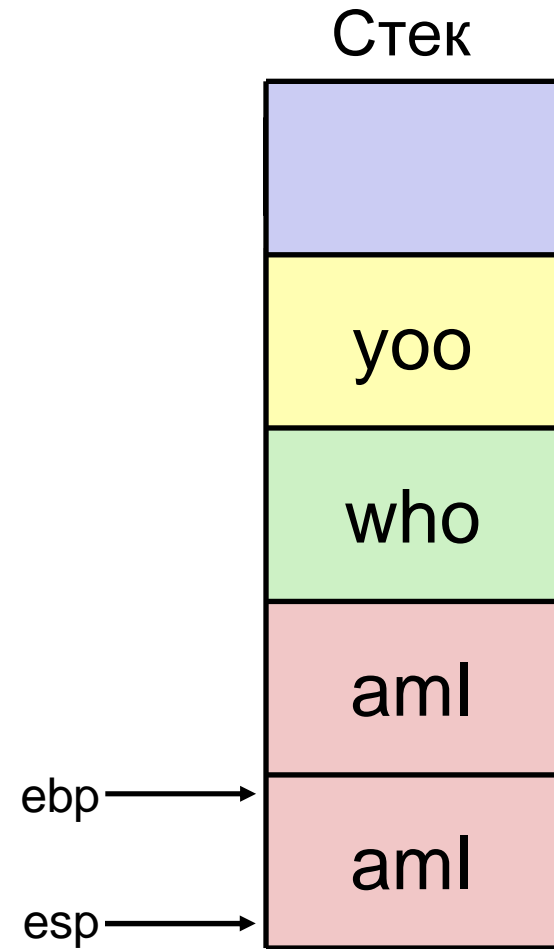
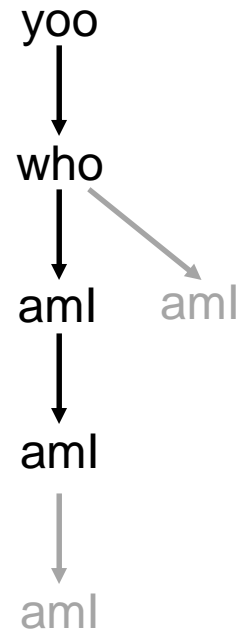
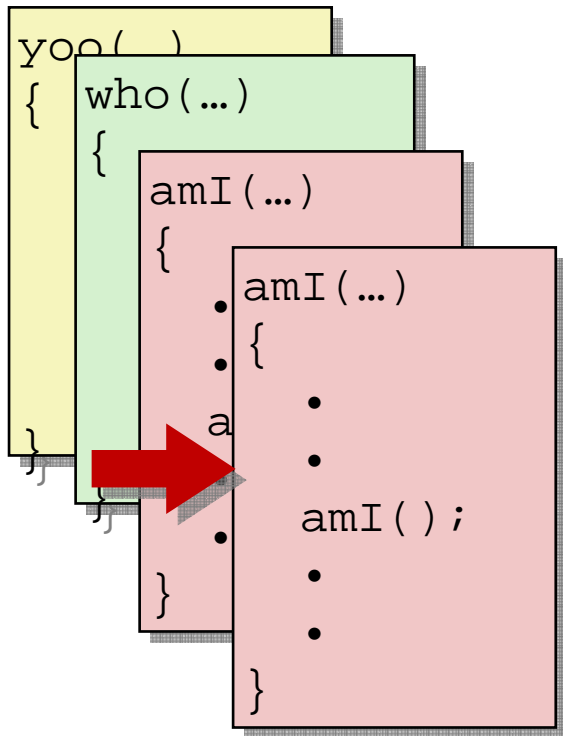


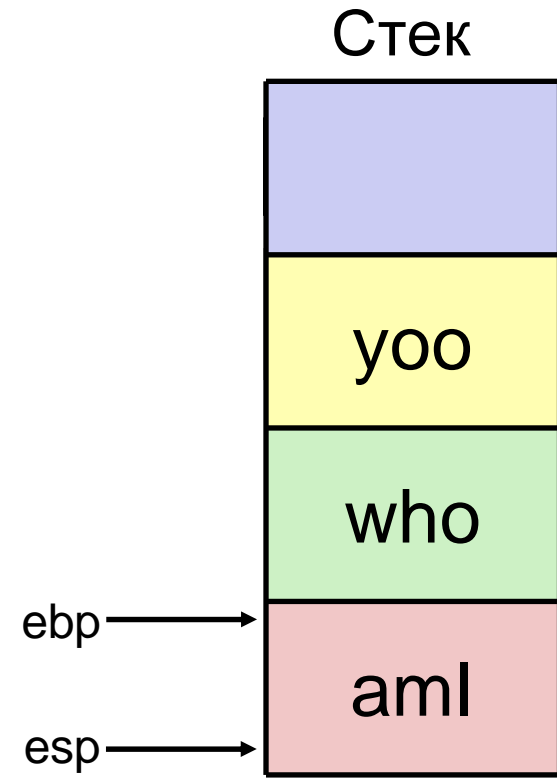
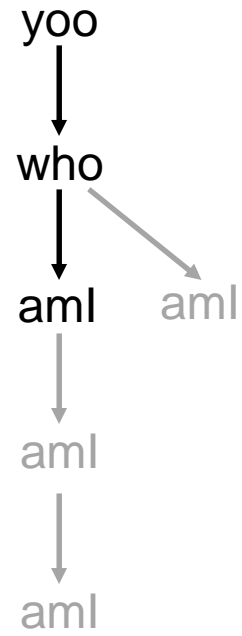
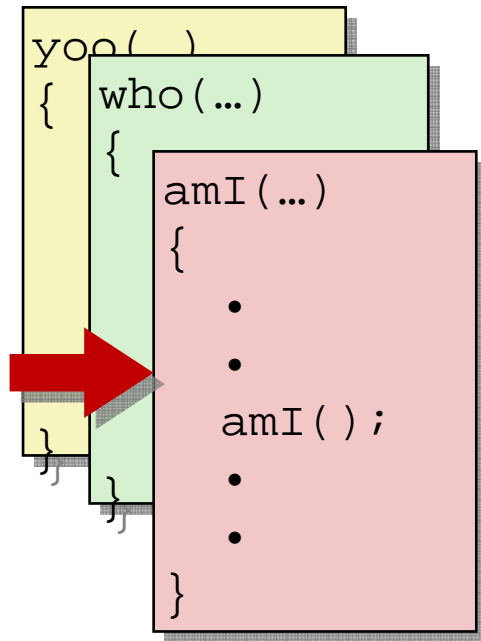


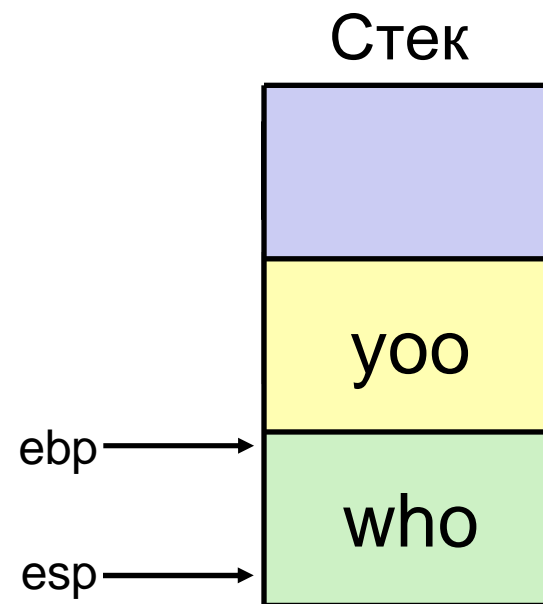
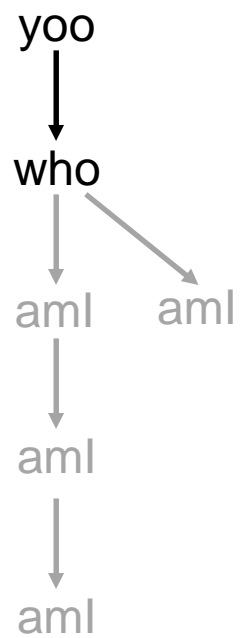
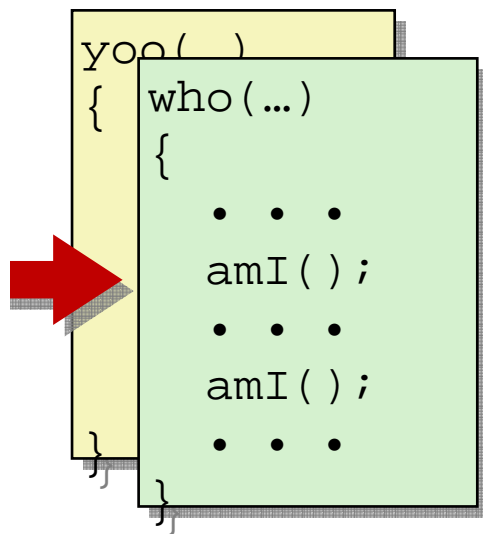


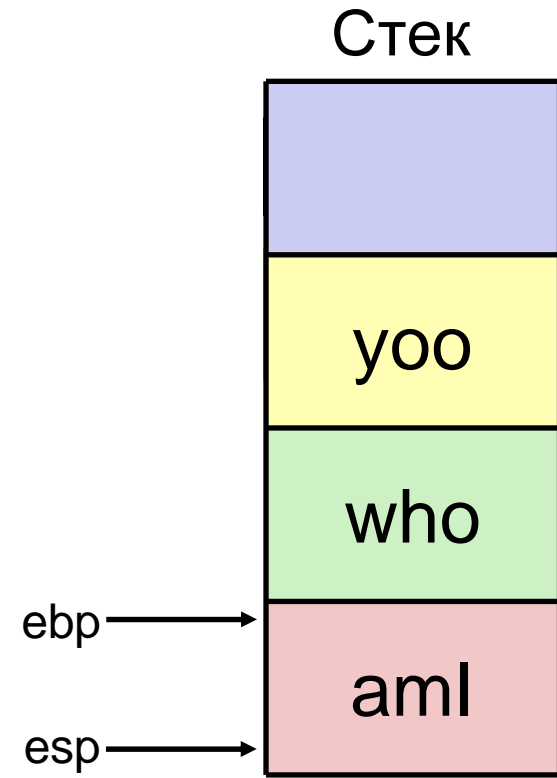
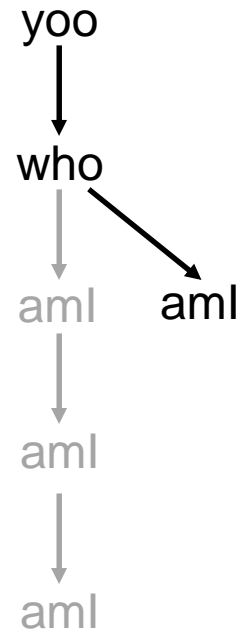
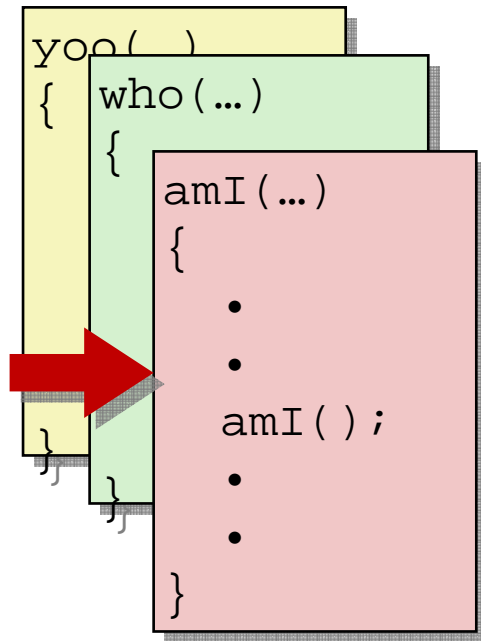


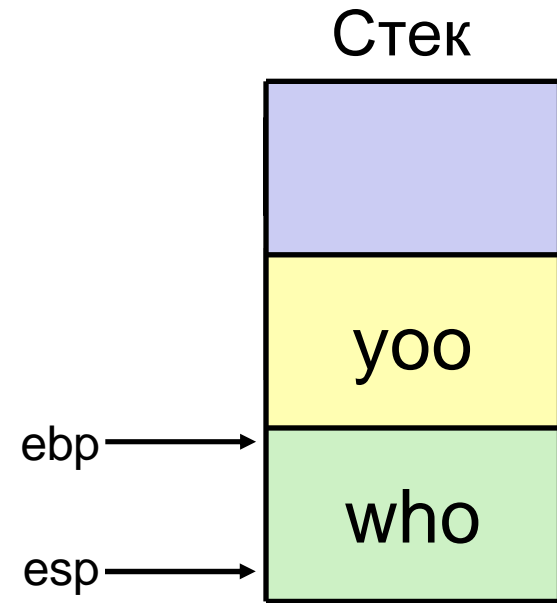
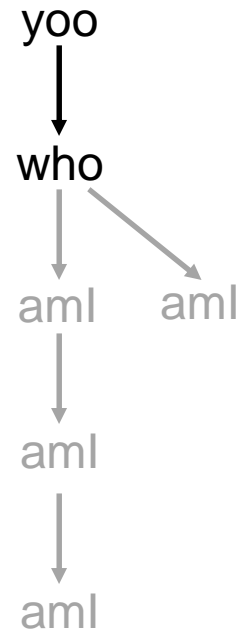
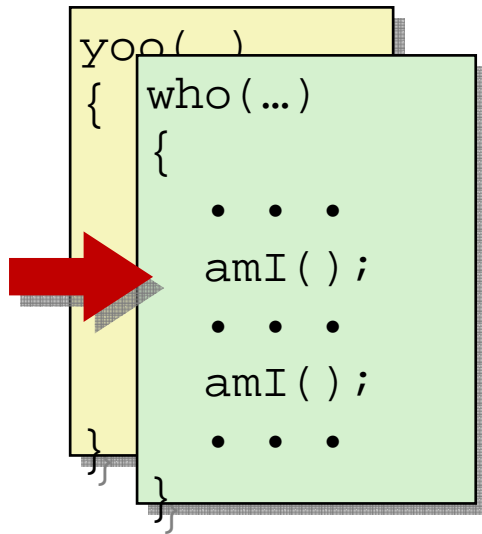


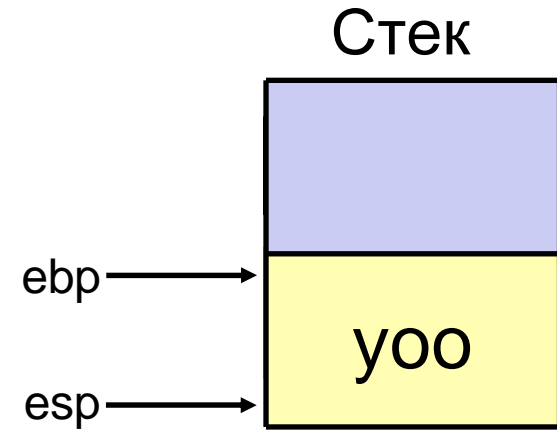
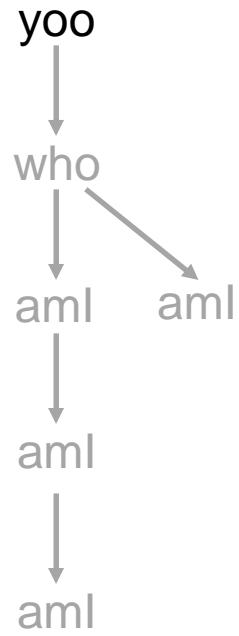
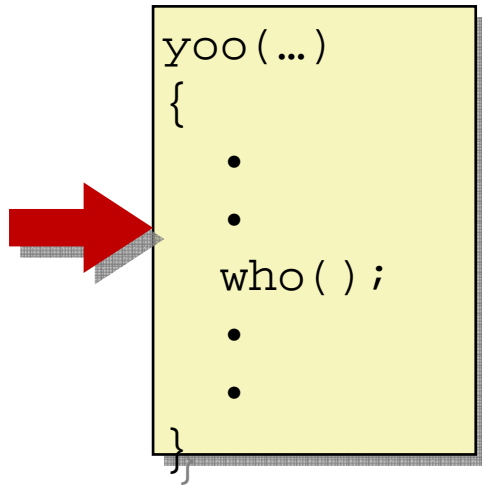












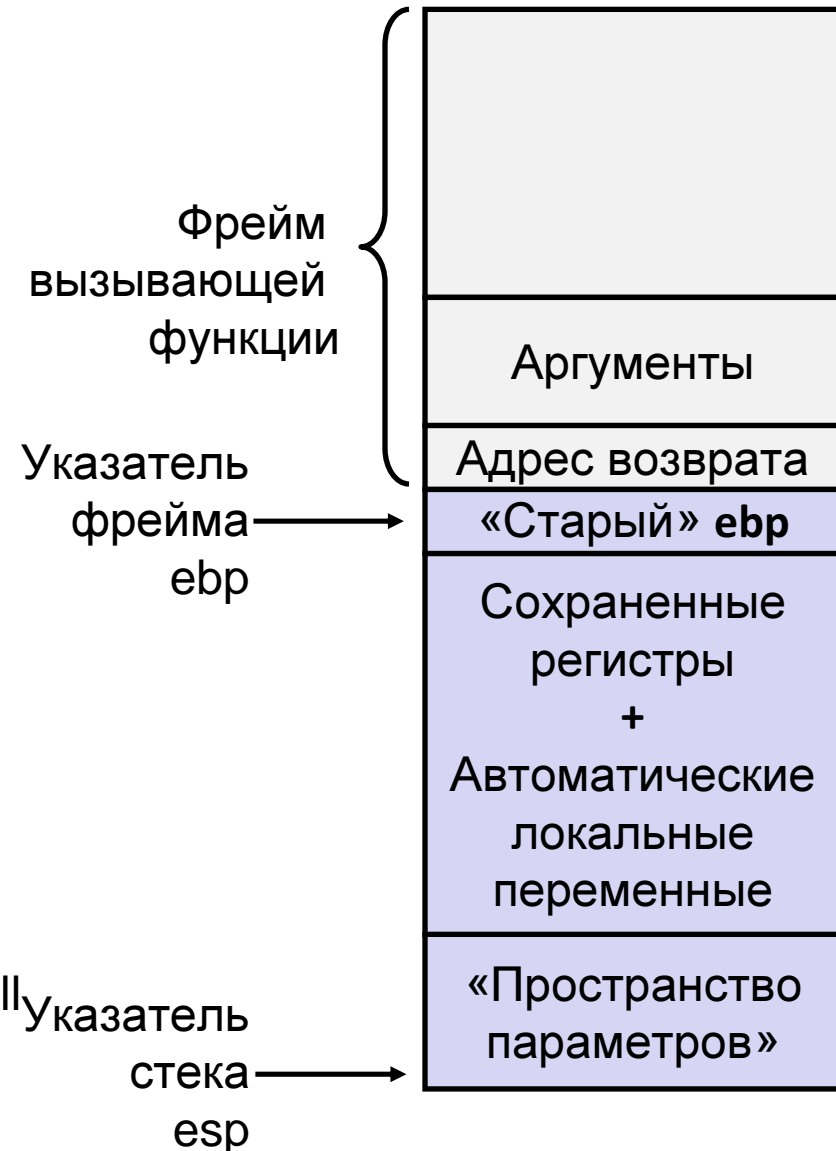
Организация фрейма в IA32/Linux

- Текущий фрейм (от «верхушки» ко «дну»)

- «Пространство параметров»: фактические параметры вызываемых функций
- Локальные переменные
- Сохраненные регистры
- Прежнее значение указателя фрейма

- Фрейм вызывающей функции

- Адрес возврата
 - Помещается на стек инструкцией `call`
- **Значения** фактических аргументов для текущего вызова




```

int group1 = 101;
int group2 = 106;

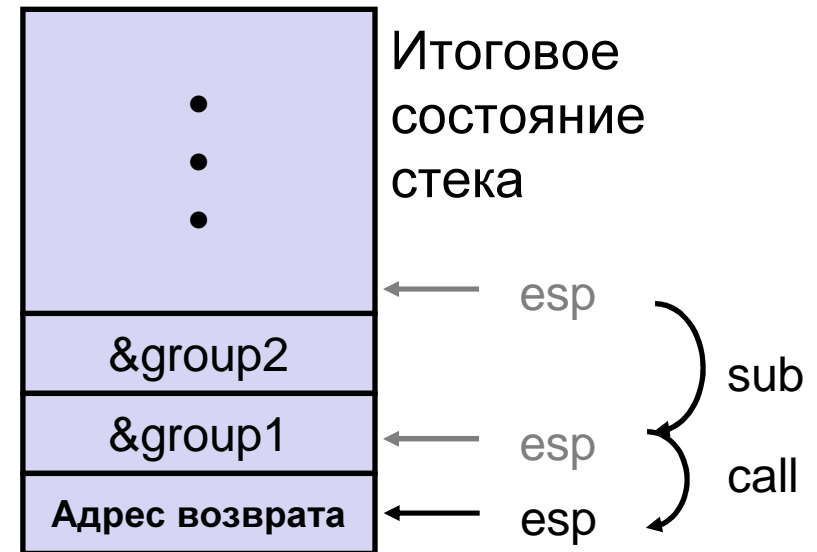
void call_swap() {
    swap(&group1, &group2);
}

```

```

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

```



Вызываем swap из call_swap

```

call_swap:
    • • •
    sub    esp, 8
    mov    dword [esp + 4], group2
    mov    dword [esp], group1
    call   swap
    • • •

```

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
push    ebp
mov     ebp, esp
push    ebx    } Пролог
```

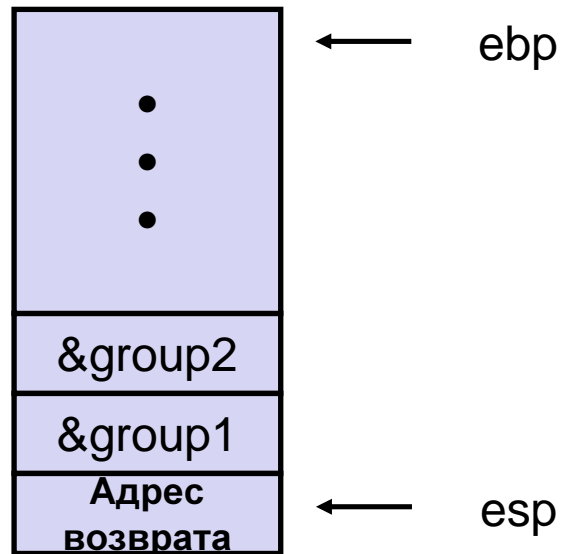
```
mov     edx, dword [ebp + 8]
mov     ecx, dword [ebp + 12]
mov     ebx, dword [edx]
mov     eax, dword [ecx]
mov     dword [edx], eax
mov     dword [ecx], ebx
```

```
pop     ebx
pop     ebp    } Эпилог
ret
```

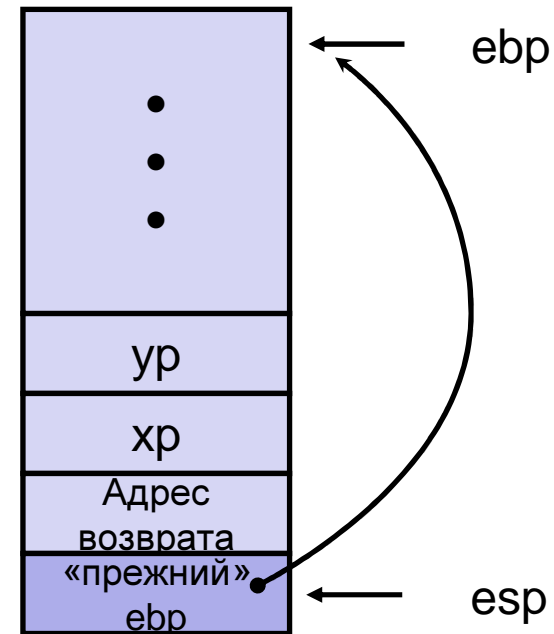
Тело
функции

Swap: как работает пролог функции. №1

Начальное
состояние стека



Текущее
состояние стека

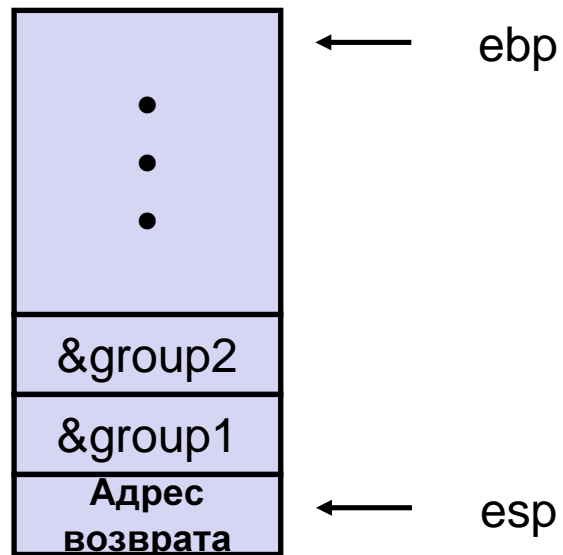


swap:

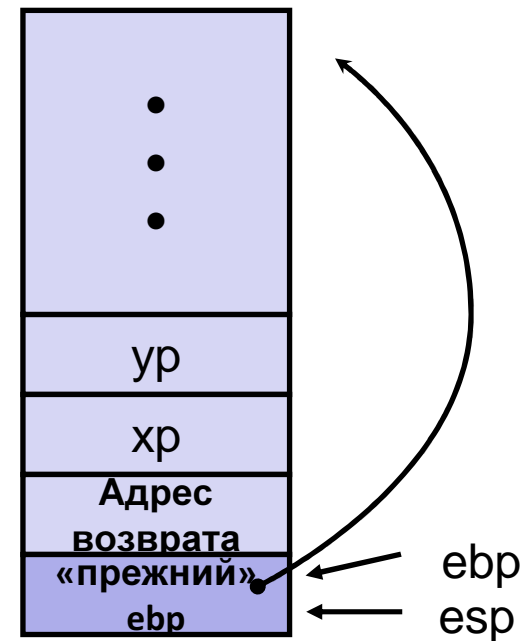
```
push ebp ; (1)  
mov ebp, esp ; (2)  
push ebx ; (3)
```

Swap: как работает пролог функции. №2

Начальное
состояние стека



Текущее
состояние стека



`swap:`

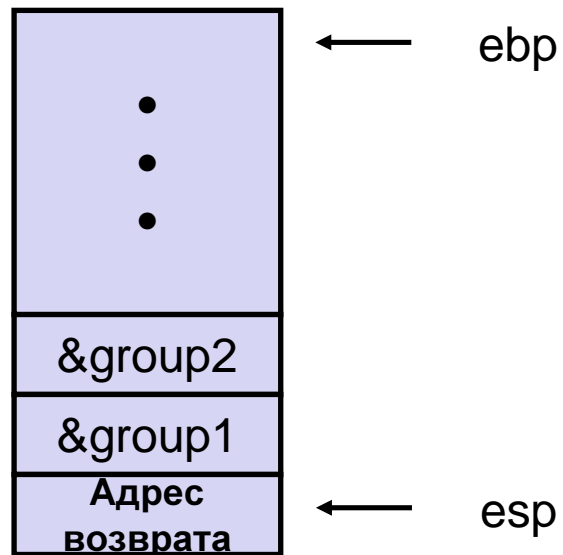
```

push ebp      ; (1)
mov  ebp, esp ; (2)
push ebx     ; (3)

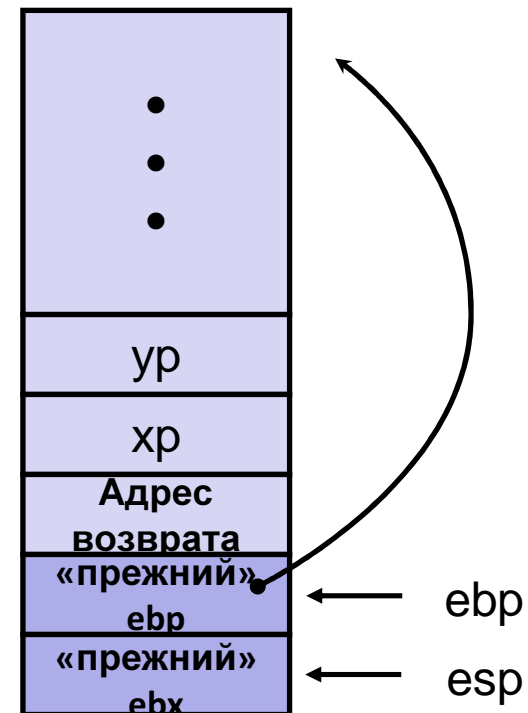
```

Swap: как работает пролог функции. №3

Начальное
состояние стека



Текущее
состояние стека



`swap:`

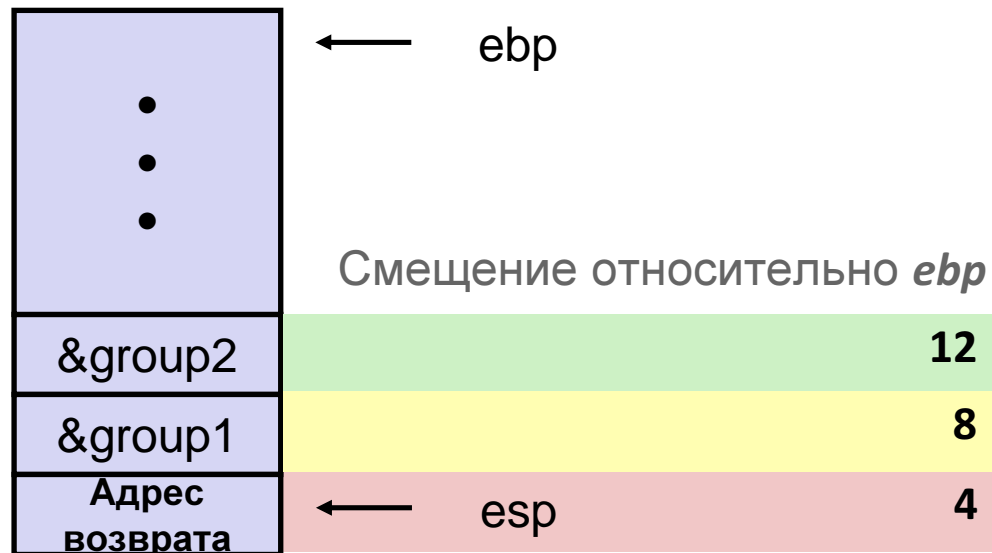
```

push ebp      ; (1)
mov  ebp, esp ; (2)
push ebx   ; (3)

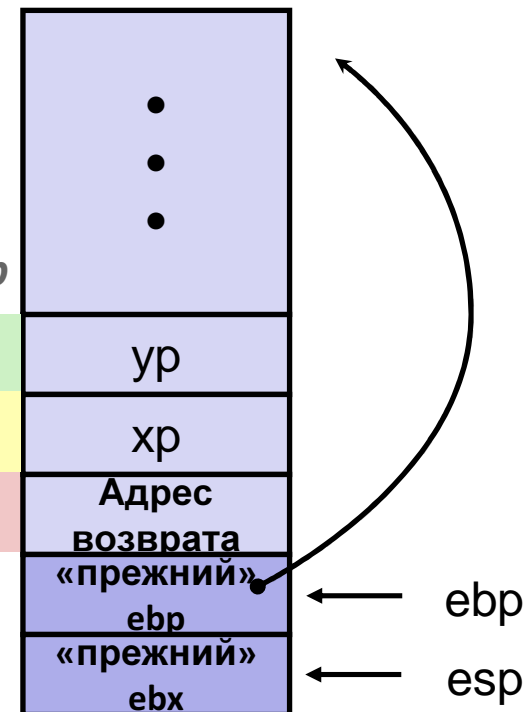
```

Тело функции swar

Начальное
состояние стека



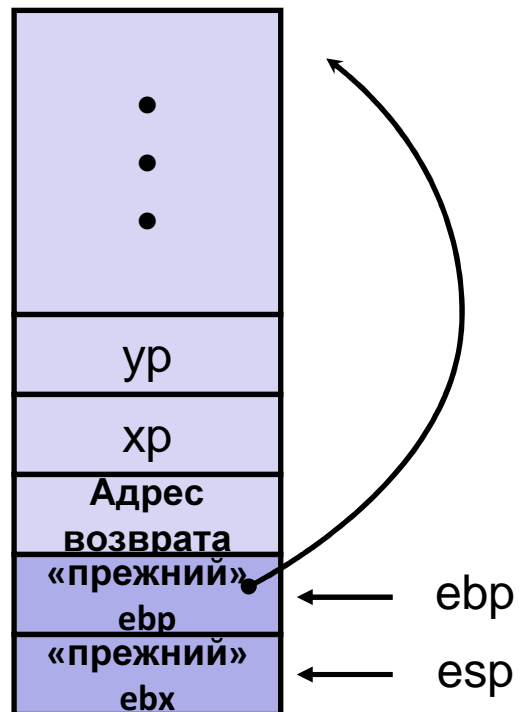
Текущее
состояние стека



```
mov edx, dword [ebp + 8] ; извлекаем параметр хр
mov ecx, dword [ebp + 12] ; извлекаем параметр ур
. . .
```

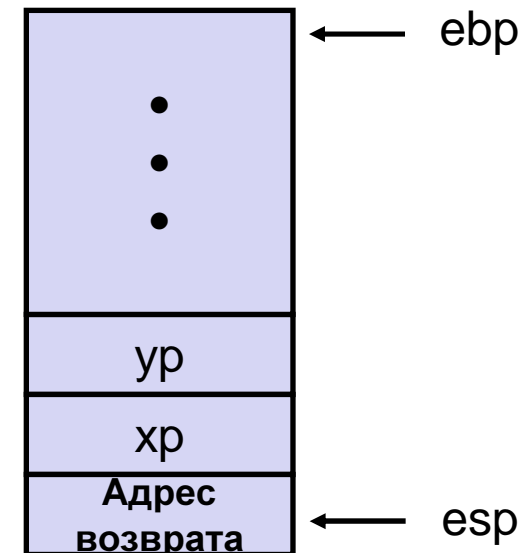
Swarp: как работает эпилог функции

Стек до эпилога



pop ebx
pop ebp

Итоговое
состояние стека

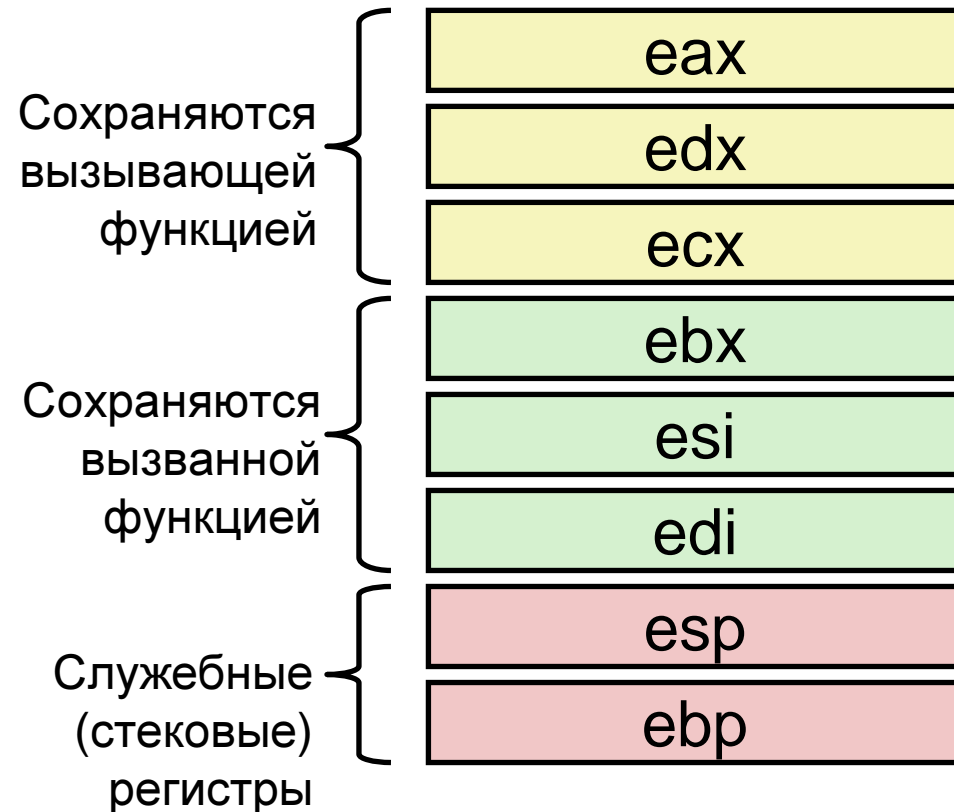


■ Что происходит

- Сохраняется и восстанавливается значение регистра ebx, ...
- ... но не eax, ecx, edx

Сохранение регистров в IA32/Linux+Windows

- `eax`, `edx`, `ecx`
 - Вызывающая функция сохраняет эти регистры перед `call`, если планирует использовать позже
- `eax`
 - Используется для возврата значения, если возвращается целый тип
- `ebx`, `esi`, `edi`
 - Вызванная функция сохраняет значения этих регистров, если планирует ими воспользоваться
- `esp`, `ebp`
 - Сохраняются вызванной функцией
 - Восстанавливаются перед выходом из функции



Рекурсивная функция

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

• Регистры

- `eax`, `edx` используются без предварительного сохранения
- `ebx` используется, сохраняется в начале и восстанавливается в конце

`pcount_r:`

```

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 4
mov     ebx, dword [ebp + 8]
mov     eax, 0
test    ebx, ebx
je     .L3
mov     eax, ebx
shr     eax, 1
mov     dword [esp], eax
call   pcount_r
mov     edx, ebx
and     edx, 1
lea    eax, [edx + eax]
.L3:
add     esp, 4
pop     ebx
pop     ebp
ret

```

Рекурсивный вызов. №1

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

pcount_r:

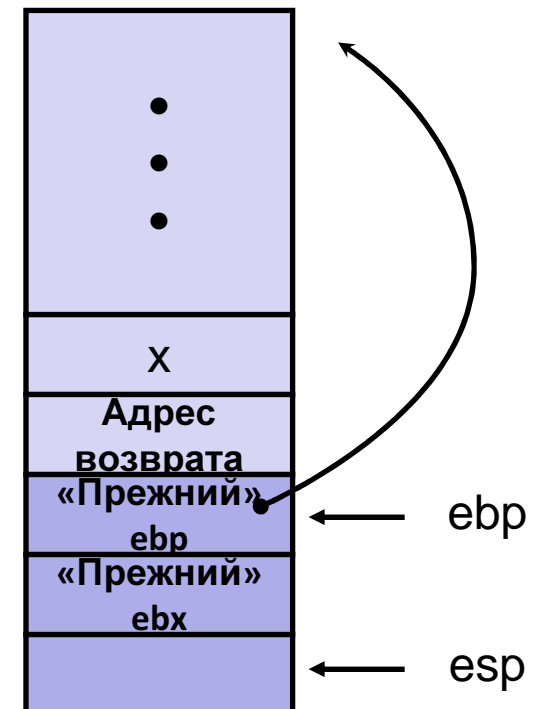
```

push  ebp
mov   ebp, esp
push  ebx
sub   esp, 4
mov   ebx, dword [ebp + 8]
    . . .

```

- Действия

- Сохраняем значение ebx на стеке
- Выделяем место для размещения аргумента вызова (рекурсивного)
- Размещаем значение x в ebx



Рекурсивный вызов. №2

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

```

    • • •
mov    eax, 0
test   ebx, ebx
je     .L3
    • • •
.L3:
    • • •
ret

```

- Действия
 - Если $x == 0$, выходим из функции
 - Регистр `eax` содержит 0

ebx x

Рекурсивный вызов. №3

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

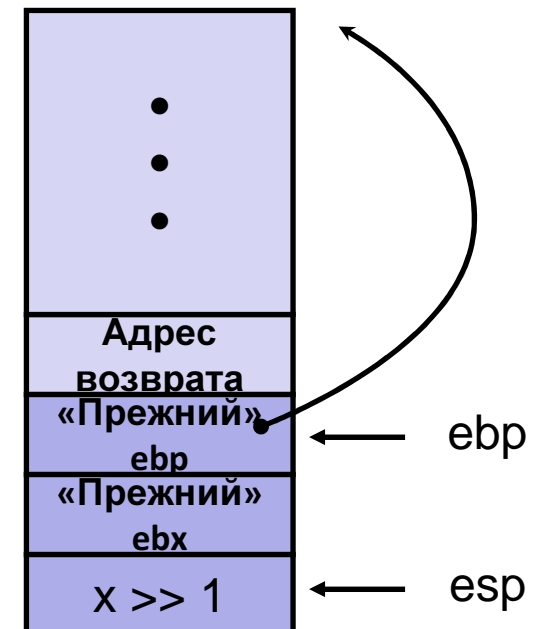
```

• • •
mov    eax, ebx
shr    eax, 1
mov    dword [esp], eax
call   pcount_r
• • •

```

- Действия
 - Сохраняем $x \gg 1$ на стеке
 - Выполняем рекурсивный вызов
- Результат
 - `eax` содержит возвращенное значение
 - `ebx` содержит неизменное значение x

`ebx` x



Рекурсивный вызов. №4

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

```

• • •
mov    edx, ebx
and    edx, 1
lea    eax, [edx + eax]
• • •

```

- Состояние регистров
 - eax содержит значение полученное от рекурсивного вызова
 - ebx содержит x
- Действия
 - Вычисляем $(x \& 1) +$ возвращенное значение
- Результат
 - Регистр eax получает результат работы функции

ebx

x

Рекурсивный вызов. №5

```

/* Рекурсивный popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

```

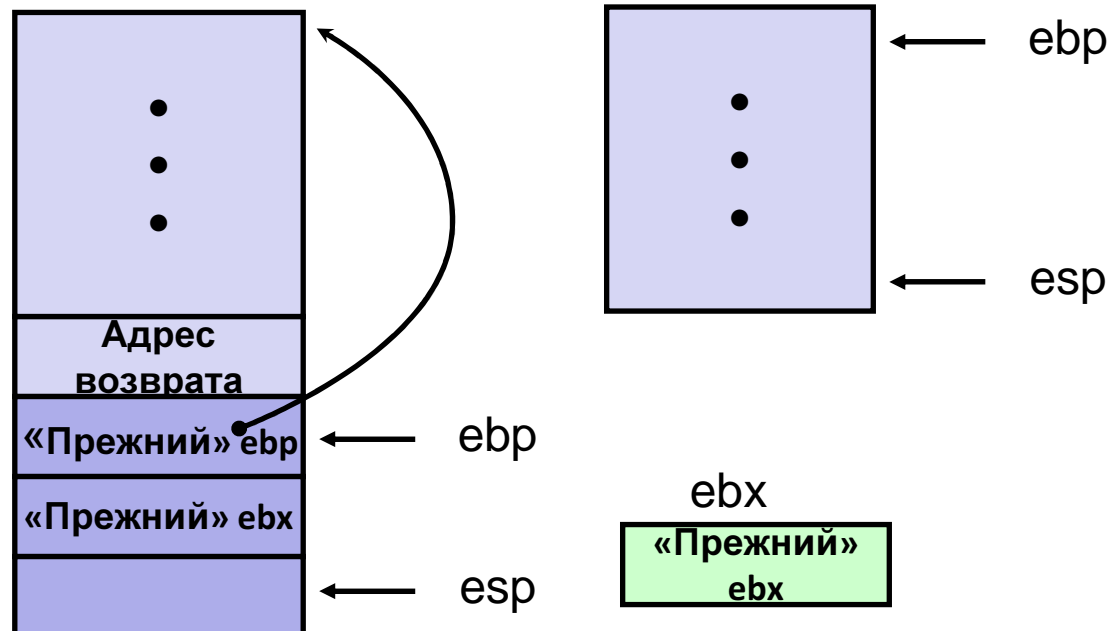
```

• • •
L3:
    add esp, 4
    pop ebx
    pop ebp
    ret

```

• Действия

- Восстанавливаем значения ebx и ebp
- Восстанавливаем esp



Рекурсия. Заключение.

- Не используются дополнительные приемы
 - Создание фреймов гарантирует, что каждый вызов располагает персональным блоком памяти
 - Хранятся регистры и локальные переменные
 - Хранится адрес возврата
 - Общее соглашение о сохранении регистров препятствует порче регистров различными вызовами
 - Стековая организация поддерживается порядком вызовов и возвратов из функций
 - Если P вызывает Q, тогда Q завершается до того, как завершится P
 - Последним пришел, первым ушел
- Аналогично при неявной рекурсии
 - P вызывает Q; Q вызывает P