Память процесса

Лекция №7
Теория языков программирования

Вопросы этой лекции

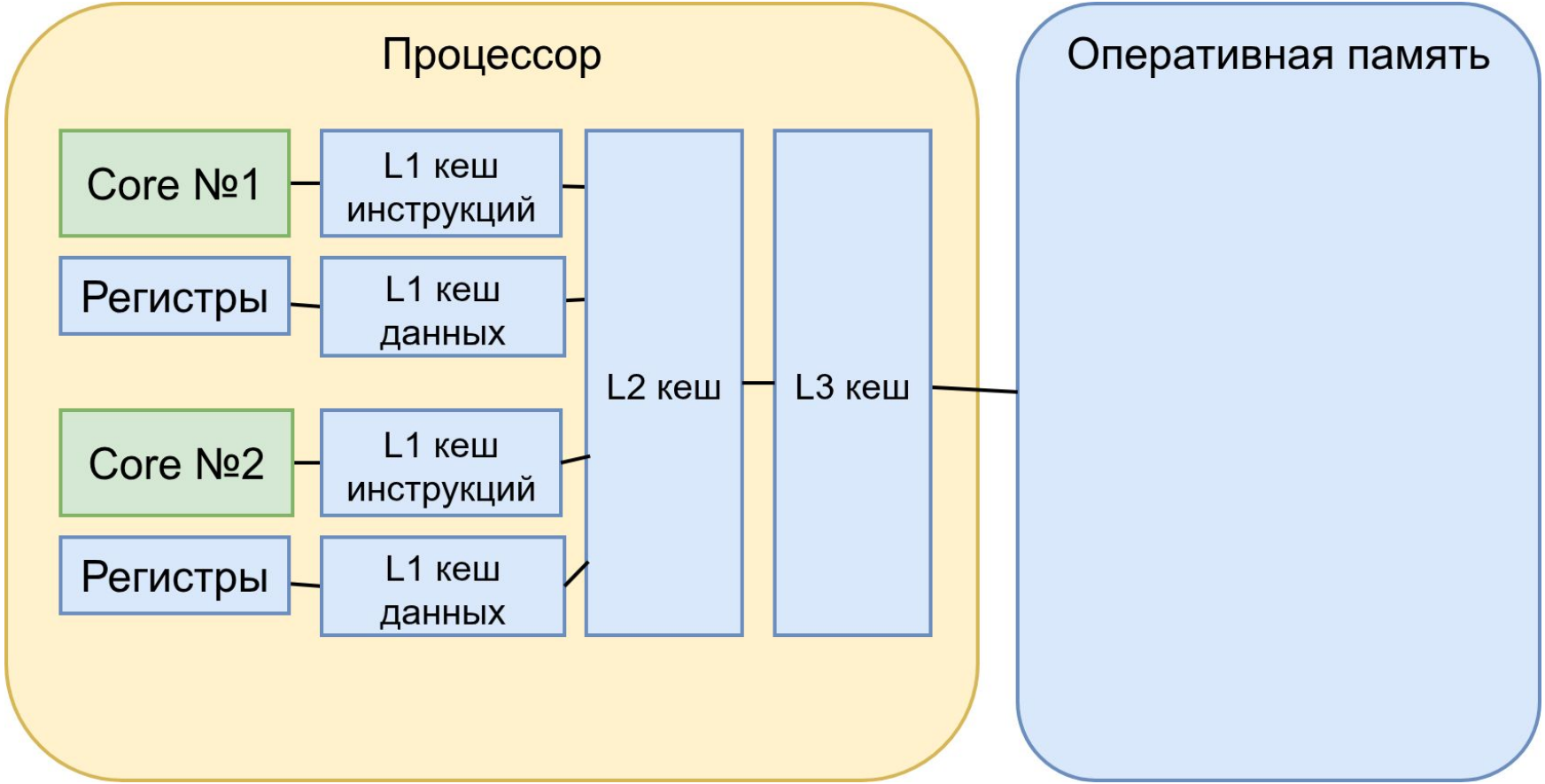
Допустим, мы пишем компилятор для AMD64 и Linux

Как устроена память процесса?

1. Где находятся локальные переменные?
2. Как функции передают параметры?
3. Где находятся данные строк?

Память, регистры, кеш

Уровни памяти для процессоров AMD / Intel



Задержки доступа к памяти (memory latency)

Уровень	Такты	Наносекунды	Общий на все ядра?
Регистры	1	0.3 нс	Нет
L1 кеш	1-3	0.3-1 нс	Нет
L2 кеш	4-10	1.3-3.3 нс	Да
L3 кеш	10-40	3.3-13.3 нс	Да
Память (ОЗУ)	60-100	20-33 нс	Да

Размер кеш-линии для Intel — 64 байта

- *16 чисел int*
- *или 8 указателей*

Контроль над уровнями памяти

Уровень	Контроль через Assembler
Регистры	Прямо указаны в инструкциях: `imull %esi, %eax`

Контроль над уровнями памяти

Уровень	Контроль через Assembler
Регистры	Прямо указаны в инструкциях: `imull %esi, %eax`
L1, L2, L3 кеш	Косвенный контроль: - prefetch - lock-флаг и протокол MESA

Контроль над уровнями памяти

Уровень	Контроль через Assembler
Регистры	Прямо указаны в инструкциях: `imull %esi, %eax`
L1, L2, L3 кеш	Косвенный контроль: - prefetch - lock-флаг и протокол MESA
Память (ОЗУ)	Адресуется через регистры и операнды инструкций `imull 64(%rsp), %r13d`

Регистры x86 (AMD/Intel 32-bit)

регистры общего назначения

%eax
%ebx
%ecx
%edx

%ebp
%esi
%edi
%esp

специальные регистры

%eip
%eflags

Регистры x86-64 (AMD/Intel 32-bit)

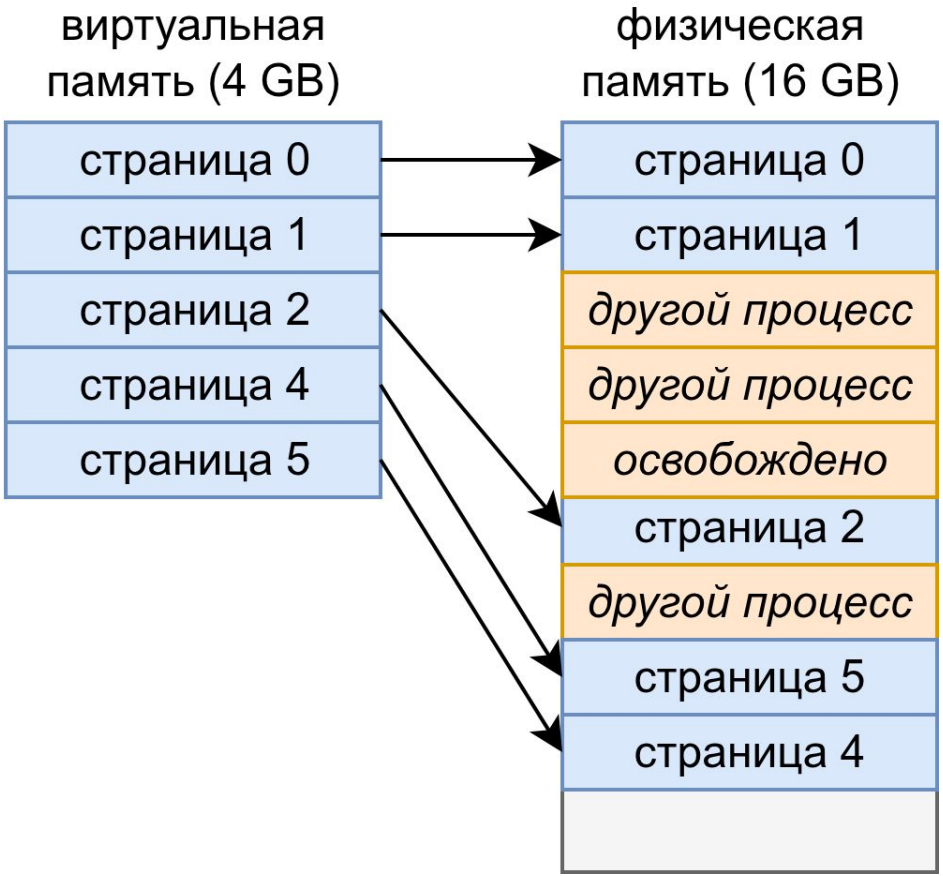
расширенные регистры x86	новые регистры	специальные регистры
%rax	%r8	%rip
%rbx	%r9	%rflags
%rcx	%r10	
%rdx	%r11	
%rbp	%r12	
%rsi	%r13	
%rdi	%r14	
%rsp	%r15	

регистры SSE:
128 бит, числа с плавающей точкой

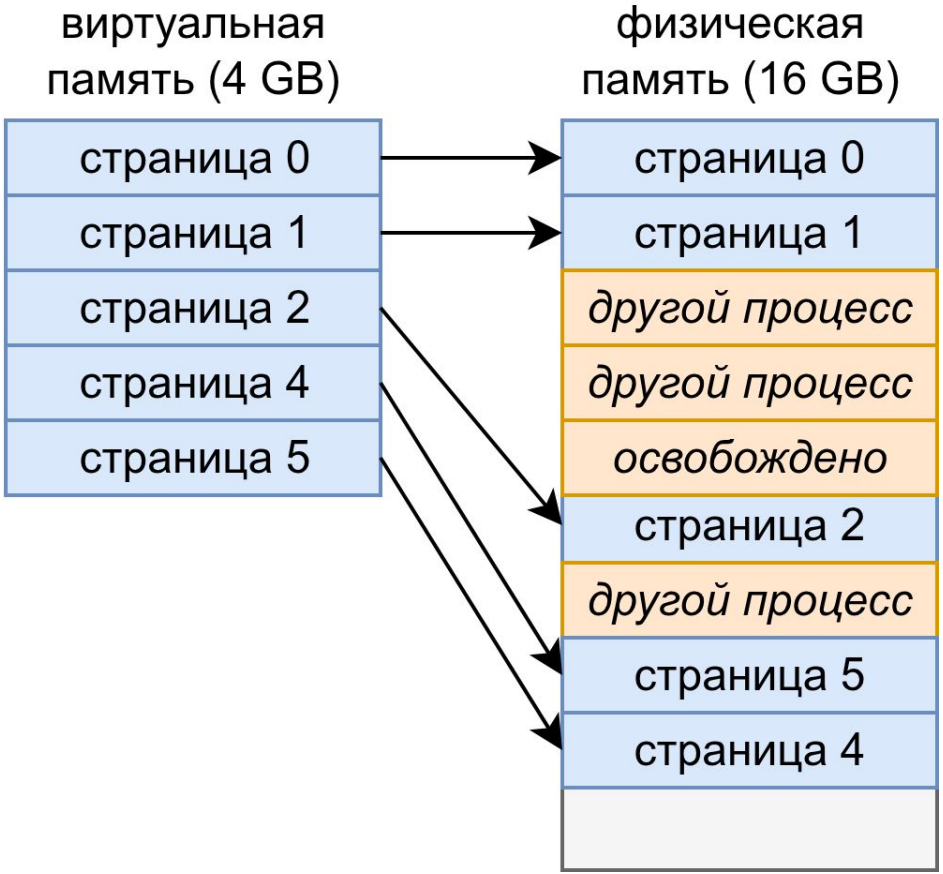
%xmm0
%xmm1
...
%xmm16

Структура памяти процесса (memory layout)

Виртуальная и физическая память

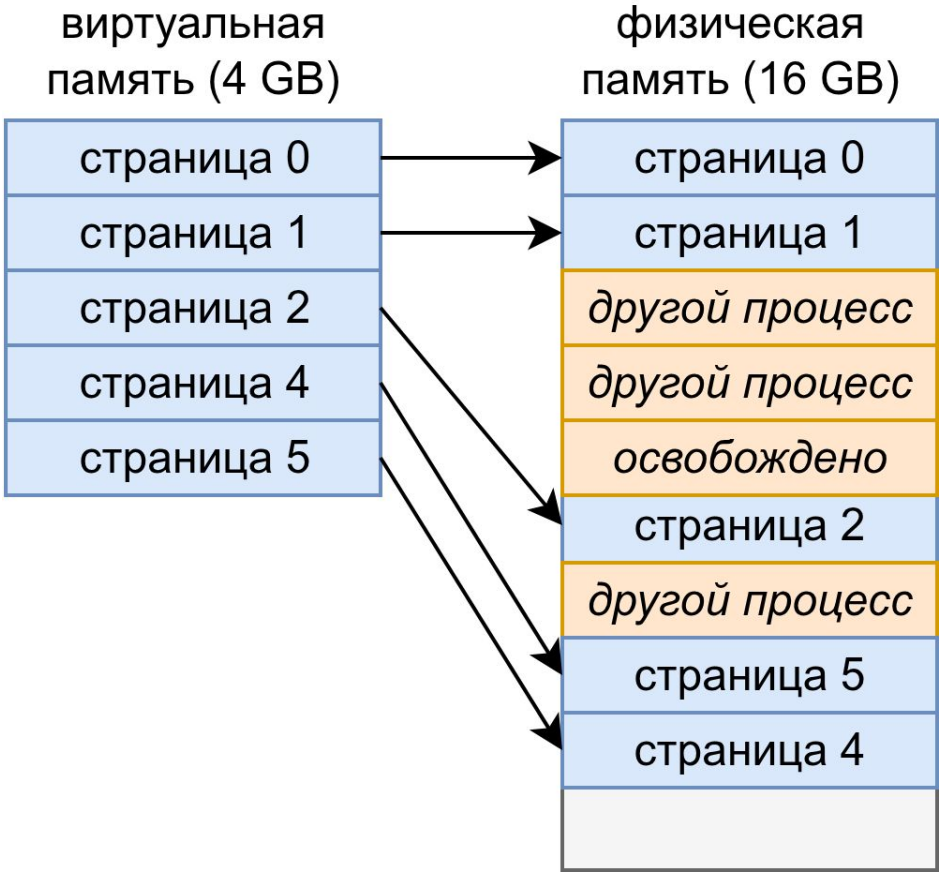


Виртуальная и физическая память



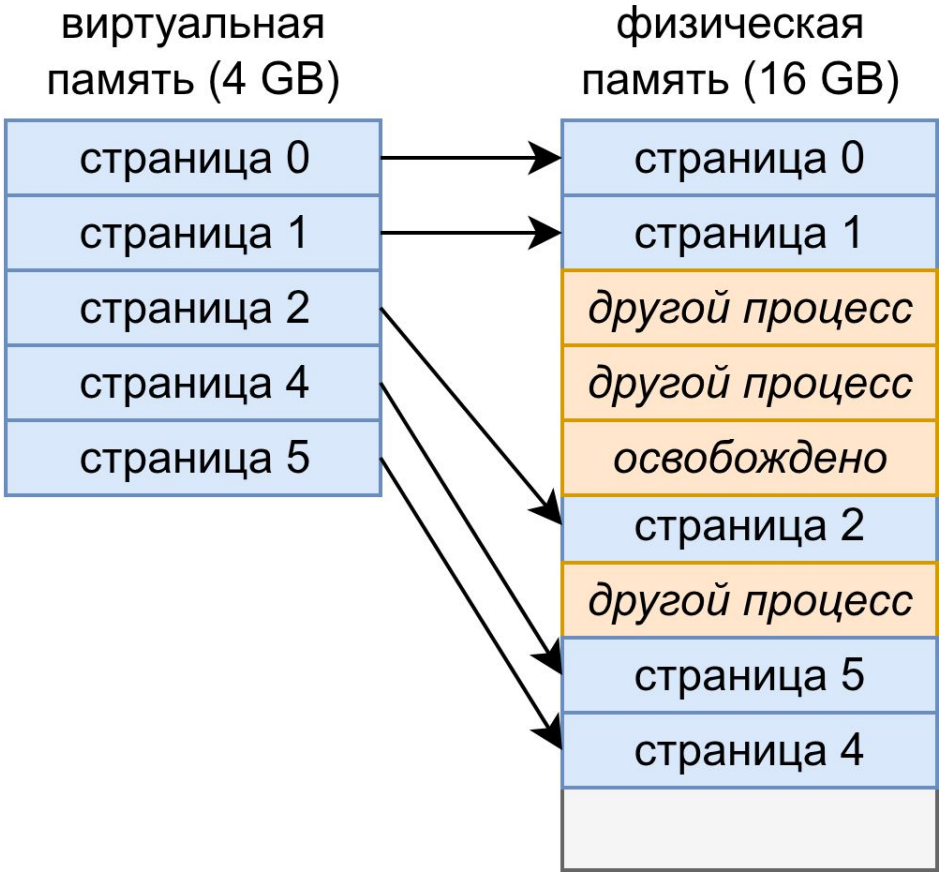
1. ОС управляет физической памятью
   - в привилегированном режиме (ring 0)

Виртуальная и физическая память



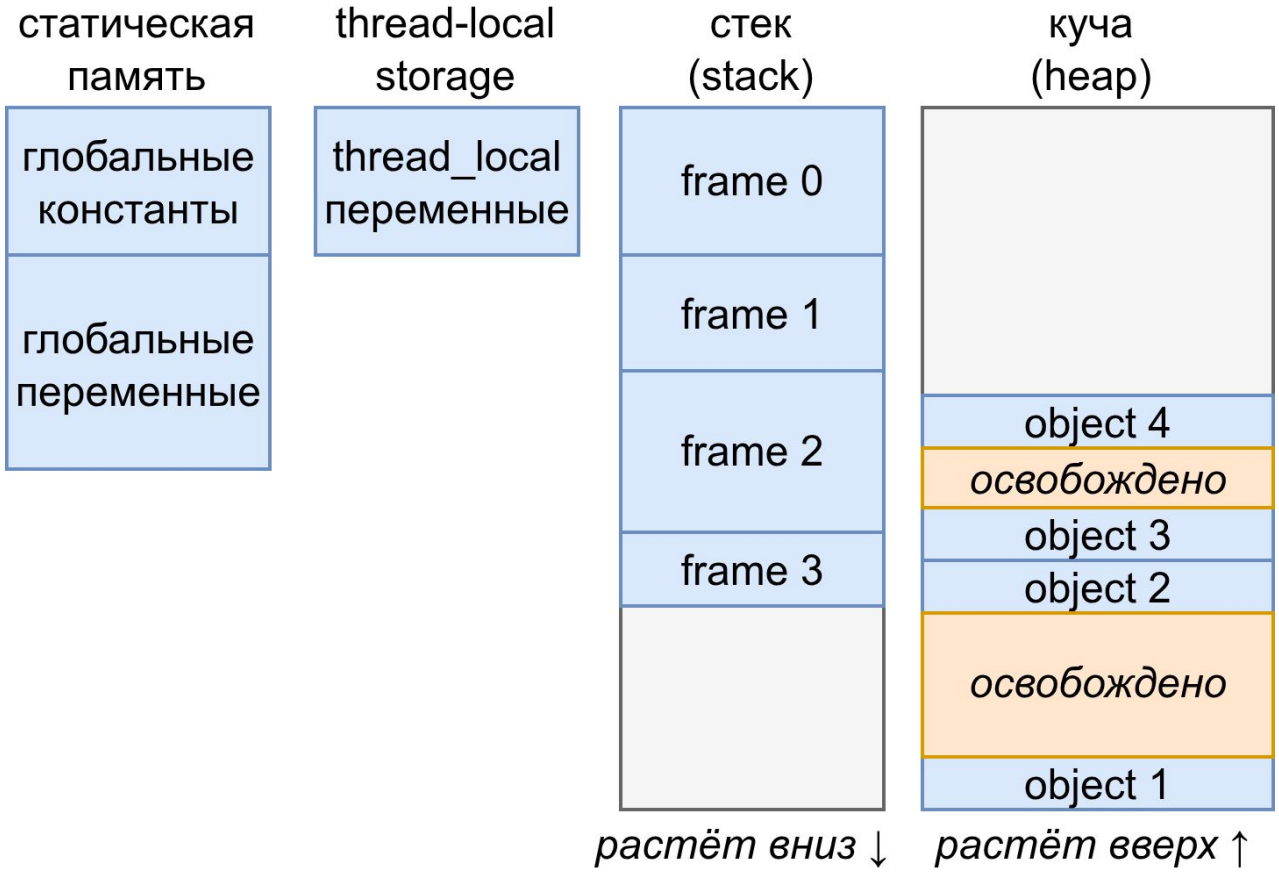
1. ОС управляет физической памятью
   - в привилегированном режиме (ring 0)
2. процесс видит только свою память
   - минимум привилегий (ring 3)
   - не считая shared memory

Виртуальная и физическая память



1. ОС управляет физической памятью
   - в привилегированном режиме (ring 0)
2. процесс видит только свою память
   - минимум привилегий (ring 3)
   - не считая shared memory
3. переключение между процессами и ОС занимает 600-1200 тактов

Сегменты памяти процесса



Сегменты памяти процесса

	С чем связан	Размер
Статическая память	процесс (process)	Неизменный
Thread-local storage	поток (thread)	Неизменный

Сегменты памяти процесса

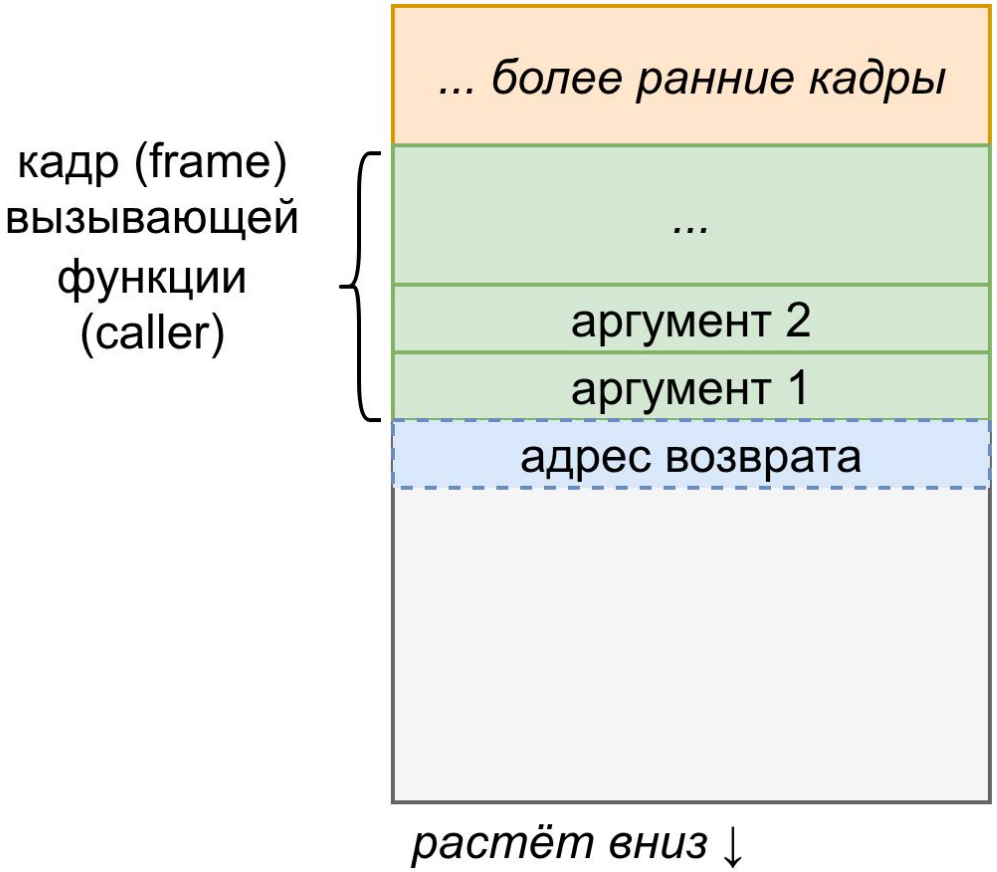
	С чем связан	Размер
Статическая память	процесс (process)	Неизменный
Thread-local storage	поток (thread)	Неизменный
Стек (stack)	поток	- Растёт вниз (адрес уменьшается) - Имеет предел роста

Сегменты памяти процесса

	С чем связан	Размер
Статическая память	процесс (process)	Неизменный
Thread-local storage	поток (thread)	Неизменный
Стек (stack)	поток	- Растёт вниз (адрес уменьшается) - Имеет предел роста
Куча (heap)	процесс	- Растёт вверх - Программа освободить память, вернув её в кучу - Подвержена фрагментации

Стек ~~программы~~ потока

Стек в архитектуре x86 (Intel 32-bit)



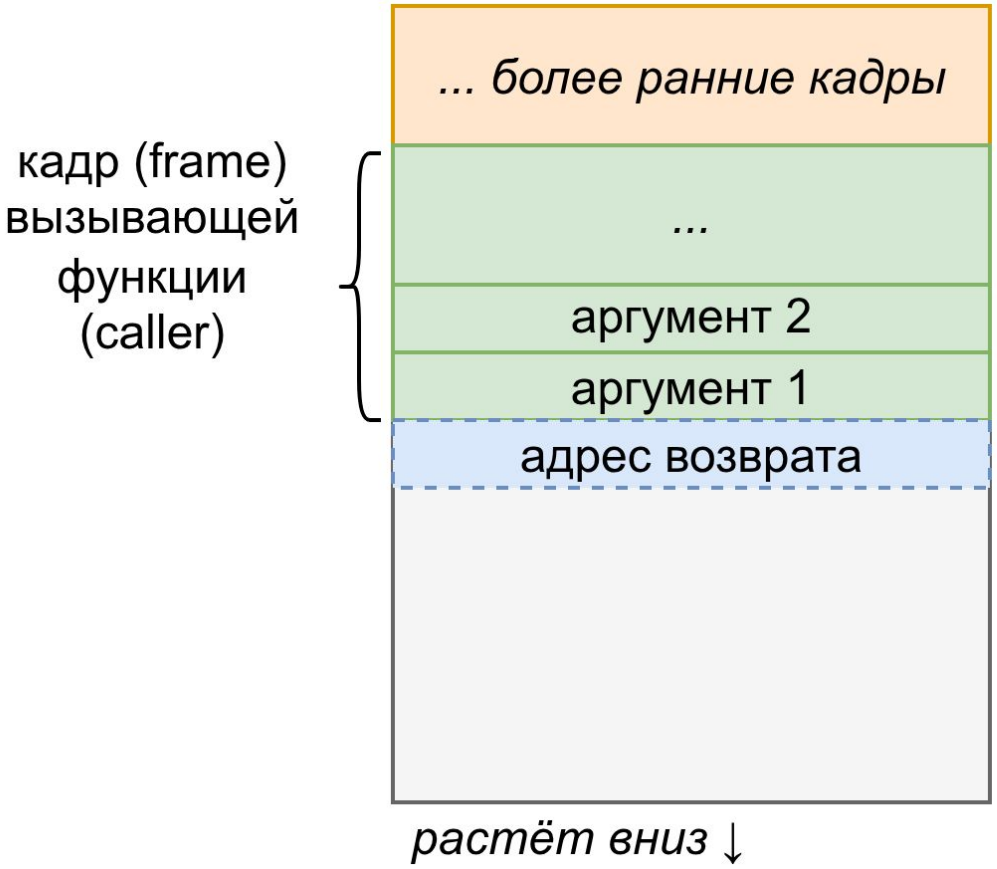
`call`

- копирует значение из указателя инструкций `%eip` на вершину стека
- уменьшает адрес в `%esp`

возвращаемое значение передаётся через регистры

- 32-битное в `%eap`,
- 64-битное в `%edx:%eax`
- вещественное в `%st0`

Стек в архитектуре x86 (Intel 32-bit)



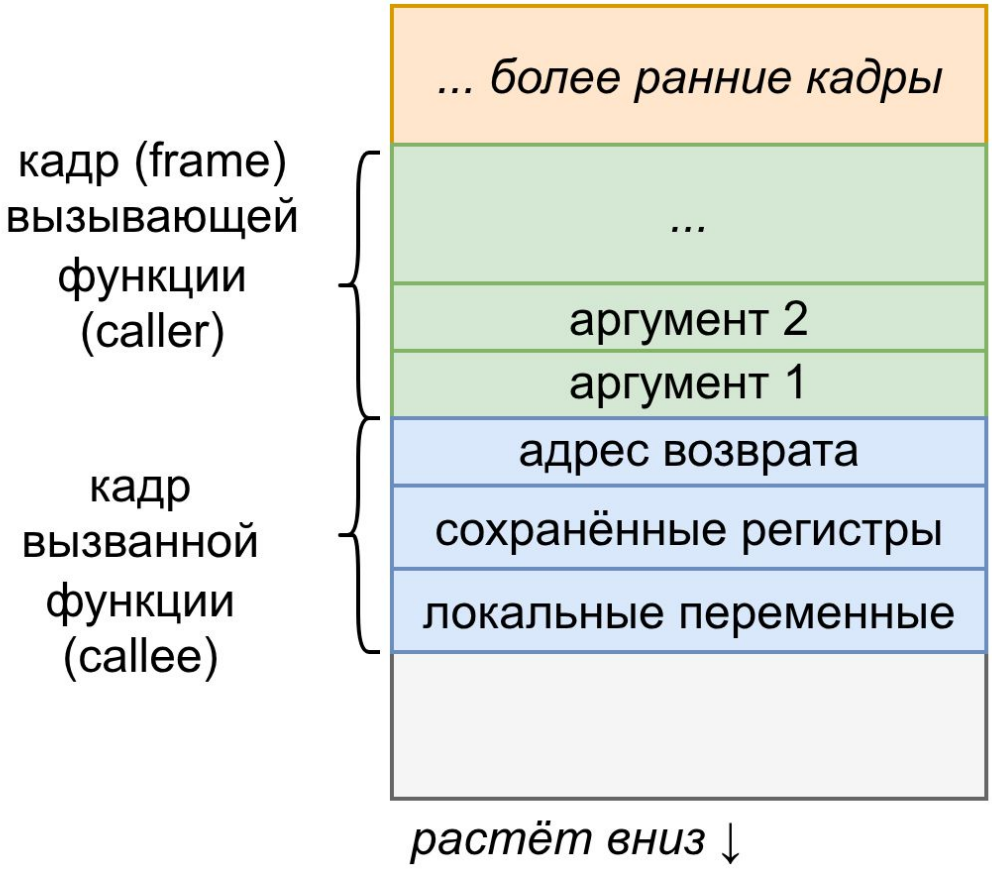
`call`

- копирует значение из указателя инструкций `%eip` на вершину стека
- уменьшает адрес в `%esp`

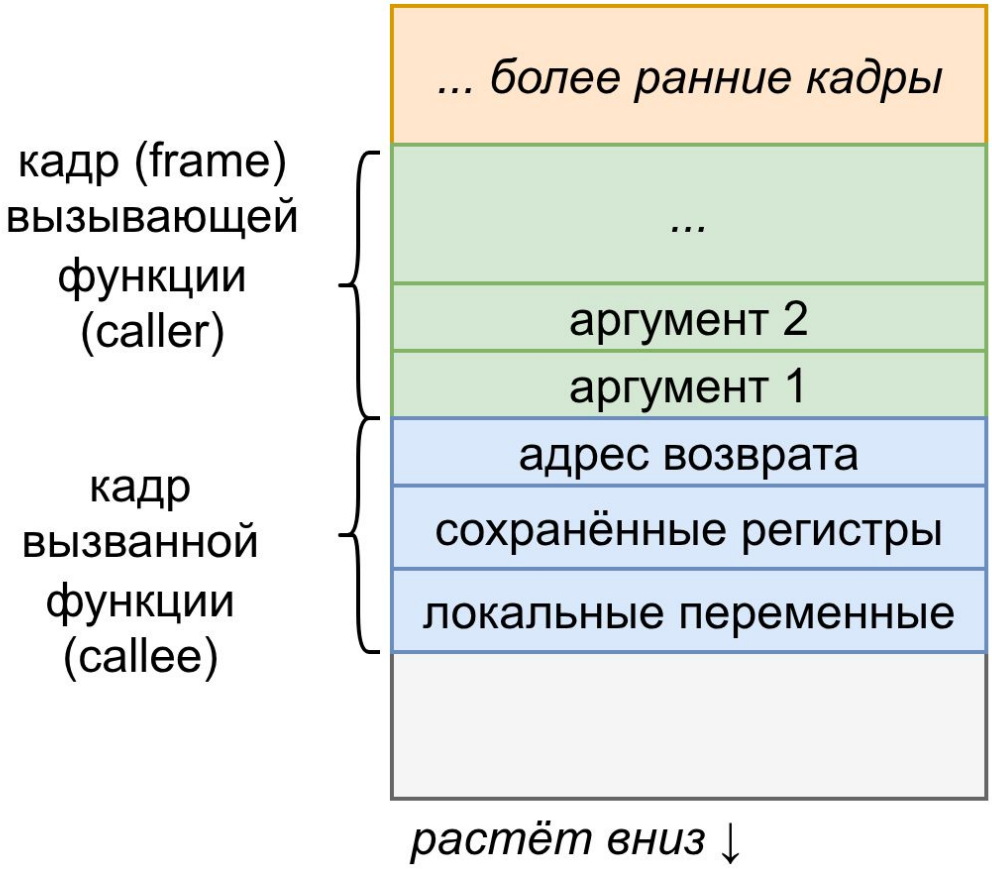
возвращаемое значение передаётся через регистры

- 32-битное в `%eap`,
- 64-битное в `%edx:%eax`
- вещественное в `%st0`

Стек в архитектуре x86 (Intel 32-bit)



Стек в архитектуре x86 (Intel 32-bit)



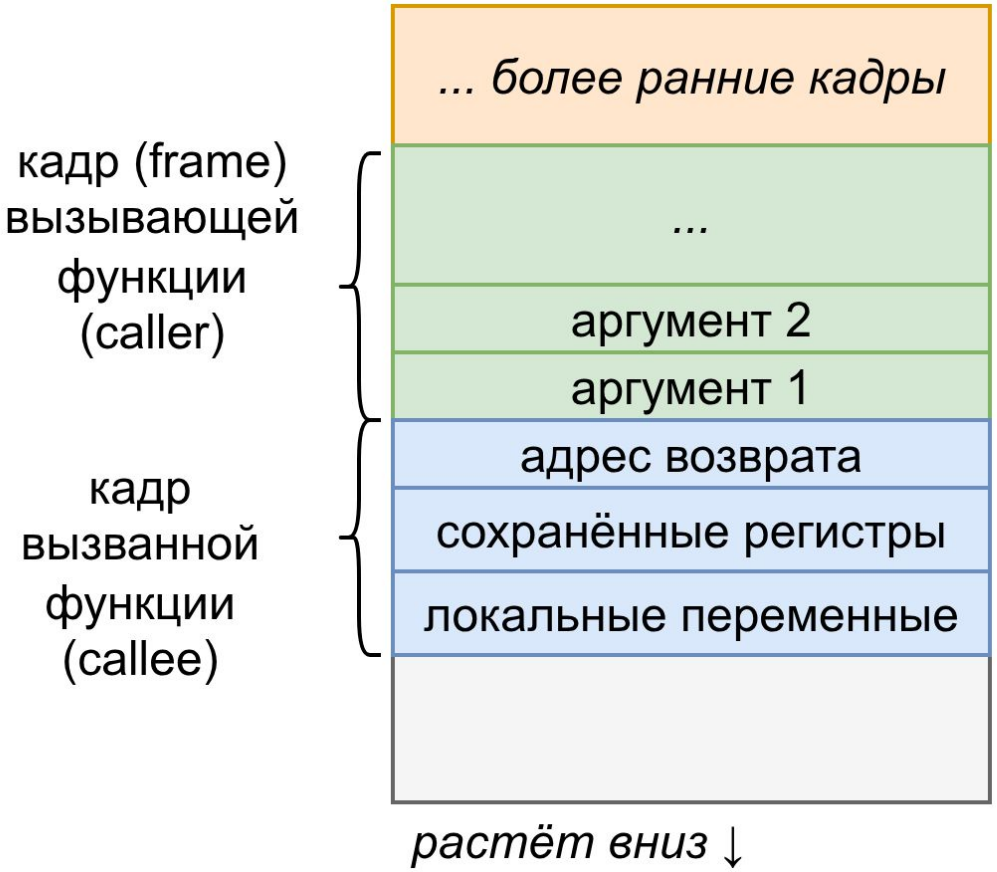
pushq %ebx

- копирует значение из %ebx на вершину
- уменьшает адрес в %esp

popq %ebx

- копирует значение из вершины в %ebx
- увеличивает адрес в %esp

Стек в архитектуре x86 (Intel 32-bit)



регистр `%ebp`

- хранит начало кадра (frame) функции
- ≥ `%esp`

регистр `%esp`

- хранит адрес вершины стека
- ≥ `%esp`

Более современные архитектуры

Общие черты:

1. Много регистров общего назначения
    - x86 было 8 — в x86-64 стало 16

Более современные архитектуры

Общие черты:

1. Много регистров общего назначения
    - x86 было 8 — в x86-64 стало 16
    - в 32-битном ARM — 16 регистров
    - в AArch64 — 31 регистр
    - в RISC-V — 16 либо 32 регистра

Более современные архитектуры

Общие черты:

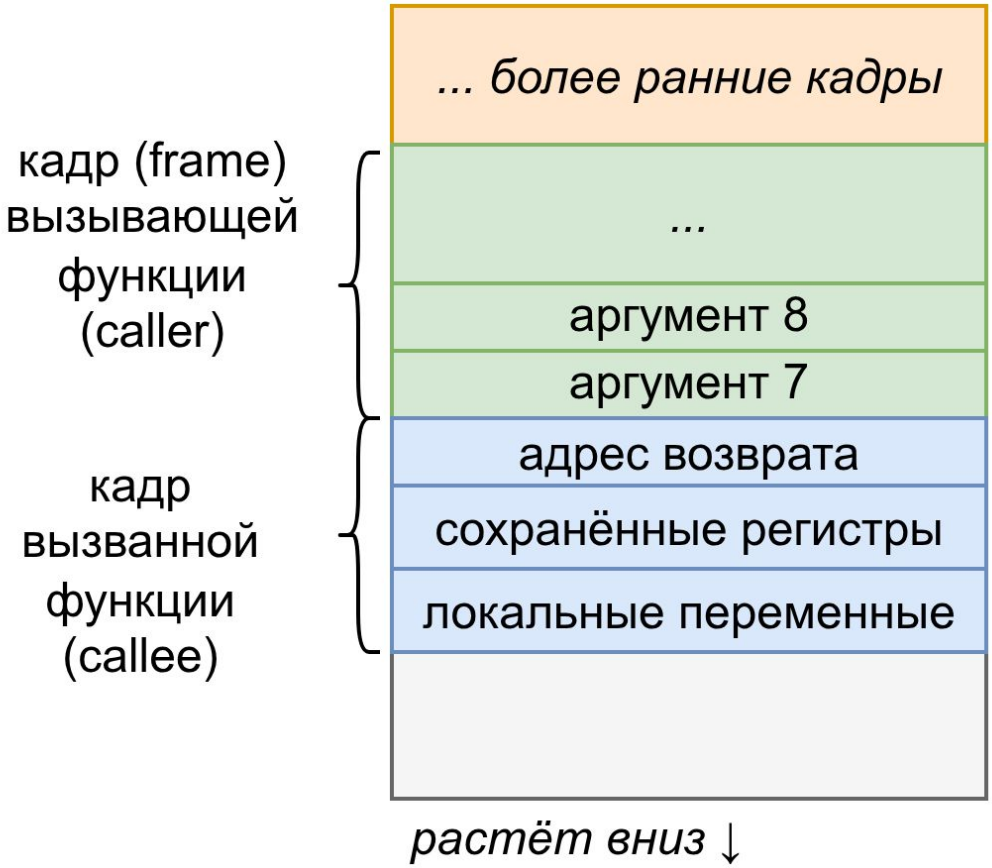
1. Много регистров общего назначения
    - x86 было 8 — в x86-64 стало 16
    - в 32-битном ARM — 16 регистров
    - в AArch64 — 31 регистр
    - в RISC-V — 16 либо 32 регистра
2. Параметры функций передаются в регистрах

Более современные архитектуры

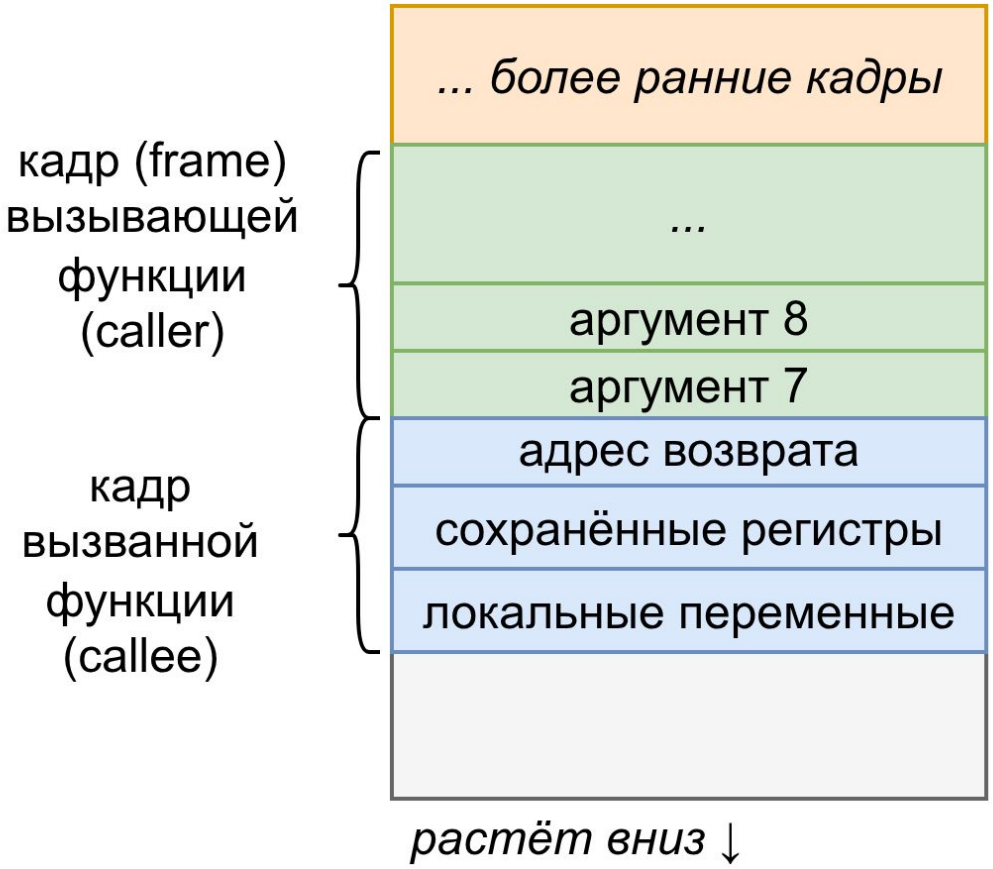
Общие черты:

1. Много регистров общего назначения
   - x86 было 8 — в x86-64 стало 16
   - в 32-битном ARM — 16 регистров
   - в AArch64 — 31 регистр
   - в RISC-V — 16 либо 32 регистра
2. Параметры функций передаются в регистрах
3. Стек используется:
   - если не хватило регистров
   - для адресов возврата

Стек в архитектуре x86-64 (64-битные Intel/AMD)



Стек в архитектуре x86-64 (64-битные Intel/AMD)



первые 6 аргументов лежат в регистрах: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

возвращаемое значение лежит в регистрах:

- 64-битное в `%rax`,
- структуры в `%rdx:%rax`
- вещественное в `%xmm0…xmm7`

Функции и переменные в LLVM IR

Распределение регистров в LLMV

- В LLVM IR бесконечное число регистров (`llvm::Value*`)

Распределение регистров в LLMV

- В LLVM IR бесконечное число регистров (`llvm::Value*`)
- Далее используется алгоритм распределения регистров (register allocation)
  - если регистров не хватает — переменная попадает в стек
  - это называется spilling

Локальные переменные в LLVM IR

Память выделяется через alloca:

```
%var.prefix = alloca ptr, align 8
%var.s1 = alloca ptr, align 8
%var.s2 = alloca ptr, align 8
```

Локальные переменные в LLVM IR

Память выделяется через alloca:

```
%var.prefix = alloca ptr, align 8
%var.s1 = alloca ptr, align 8
%var.s2 = alloca ptr, align 8
```

В компиляторе:

```
llvm::Twine name("var.", d.name());
llvm::Type* type = mapType(d.resultType());
llvm::Value* variable = _builder.CreateAlloca(type, nullptr, name);
```

Локальные переменные в LLVM IR

Память выделяется через alloca:

```
%var.prefix = alloca ptr, align 8
%var.s1 = alloca ptr, align 8
%var.s2 = alloca ptr, align 8
```

Вопрос: как убрать лишние alloca, поместив переменную в регистр?

Локальные переменные в LLVM IR

Память выделяется через alloca:

```
%var.prefix = alloca ptr, align 8
%var.s1 = alloca ptr, align 8
%var.s2 = alloca ptr, align 8
```

Вопрос: как убрать лишние alloca, поместив переменную в регистр?

Ответ: LLVM сам сделает это (оптимизация mem2reg)

Функции в LLVM IR

LLVM IR абстрагирует соглашения о вызове:

```
define internal ptr @concat_with_space(ptr %s1, ptr %s2) {
entry:
 %var.prefix = alloca ptr, align 8
 // ...
 ret ptr %concat.1
}
```

Функции в LLVM IR

LLVM IR абстрагирует соглашения о вызове:

```
define internal ptr @concat_with_space(ptr %s1, ptr %s2) {
entry:
 %var.prefix = alloca ptr, align 8
 // ...
 ret ptr %concat.1
}
```

Вопрос: что значит internal и зачем он нужен?

Функции в LLVM IR

LLVM IR абстрагирует соглашения о вызове:

```
define internal ptr @concat_with_space(ptr %s1, ptr %s2) {
entry:
 %var.prefix = alloca ptr, align 8
 // ...
 ret ptr %concat.1
}
```

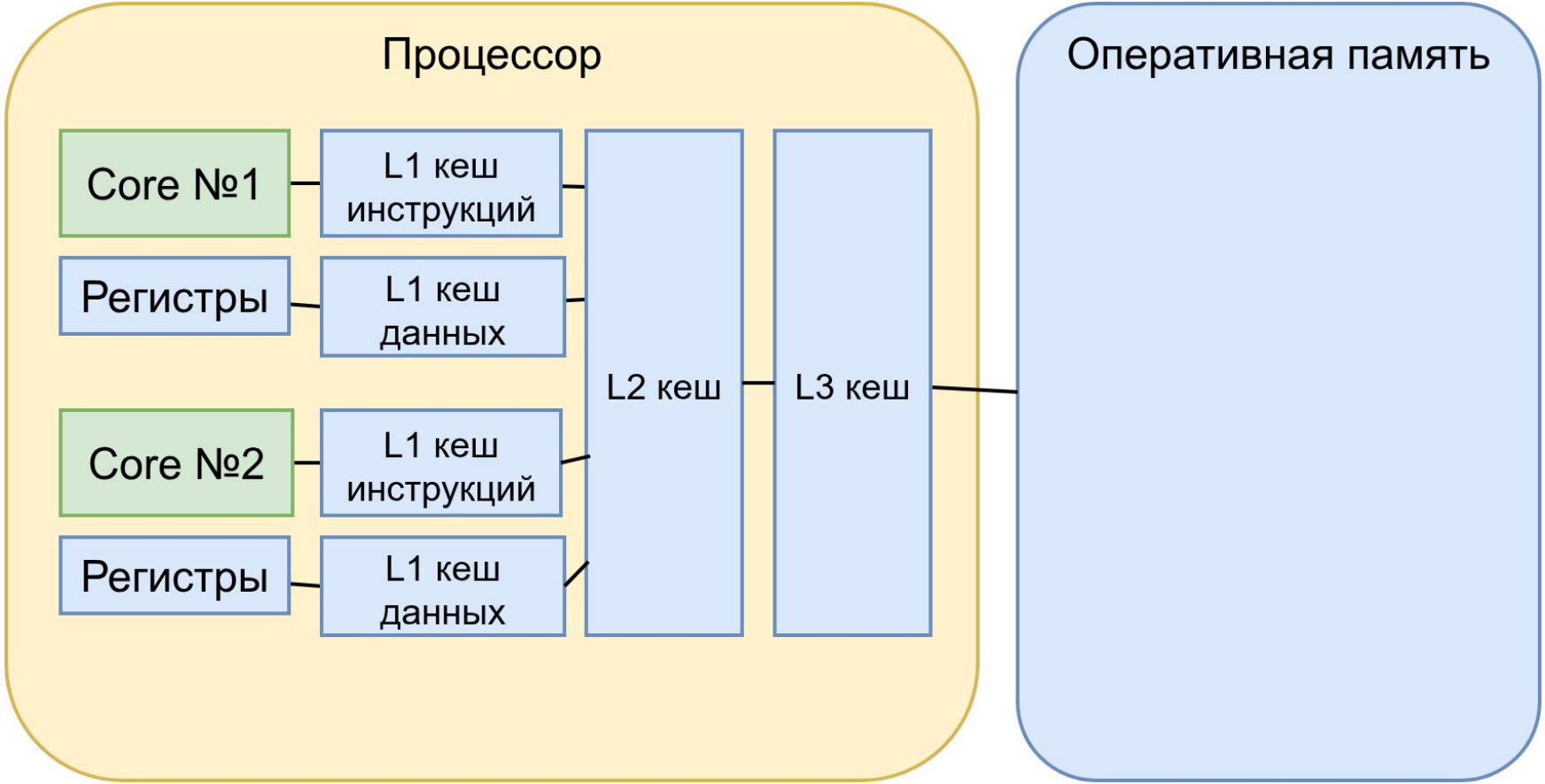
Вопрос: что значит internal и зачем он нужен?

Ответ:

- при компоновке internal функция не видна другим модулям
- это позволяет оптимизацию inlining до компоновки

Подытожим

Уровни памяти для процессоров AMD / Intel



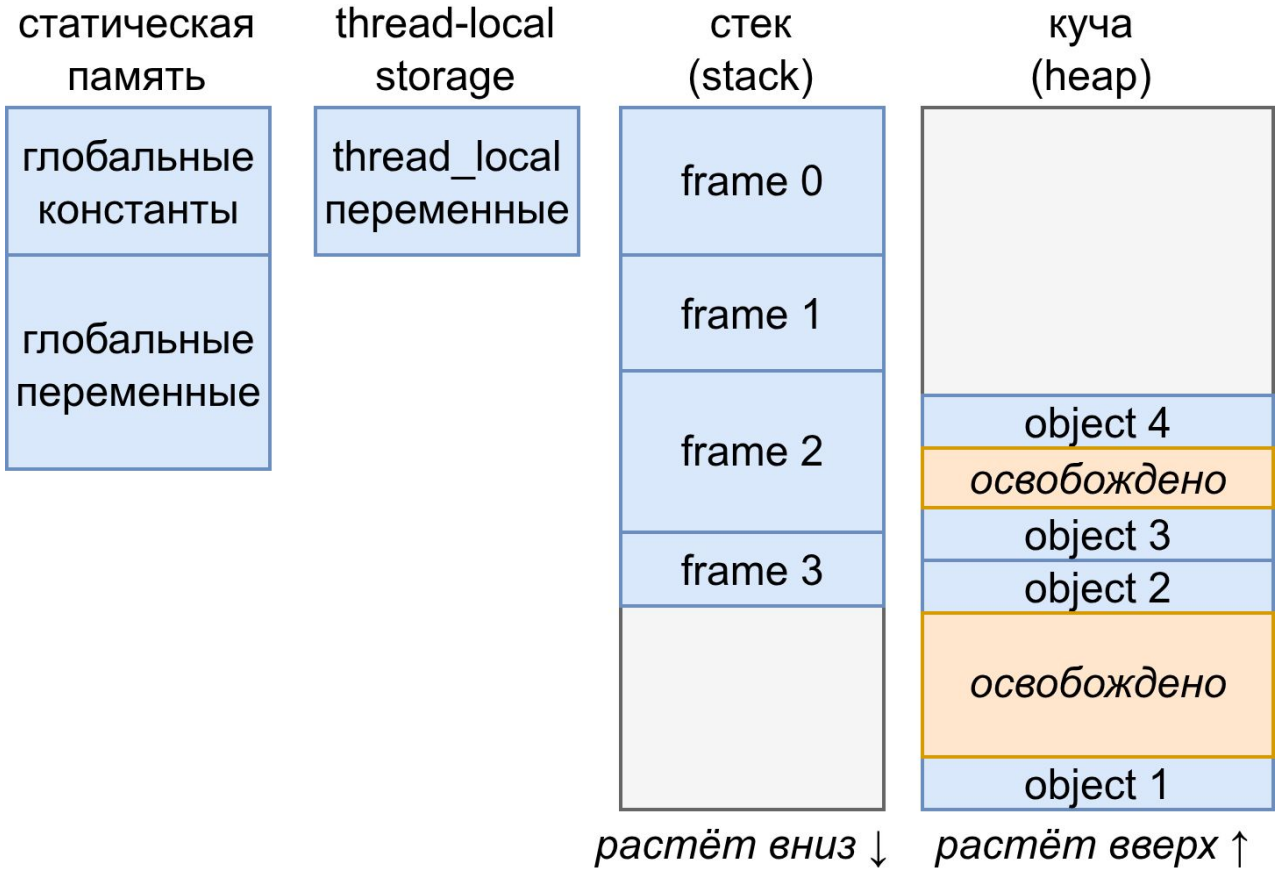
Регистры x86-64 (AMD/Intel 32-bit)

расширенные регистры x86	новые регистры	специальные регистры
%rax	%r8	%rip
%rbx	%r9	%rflags
%rcx	%r10	
%rdx	%r11	
%rbp	%r12	
%rsi	%r13	
%rdi	%r14	
%rsp	%r15	

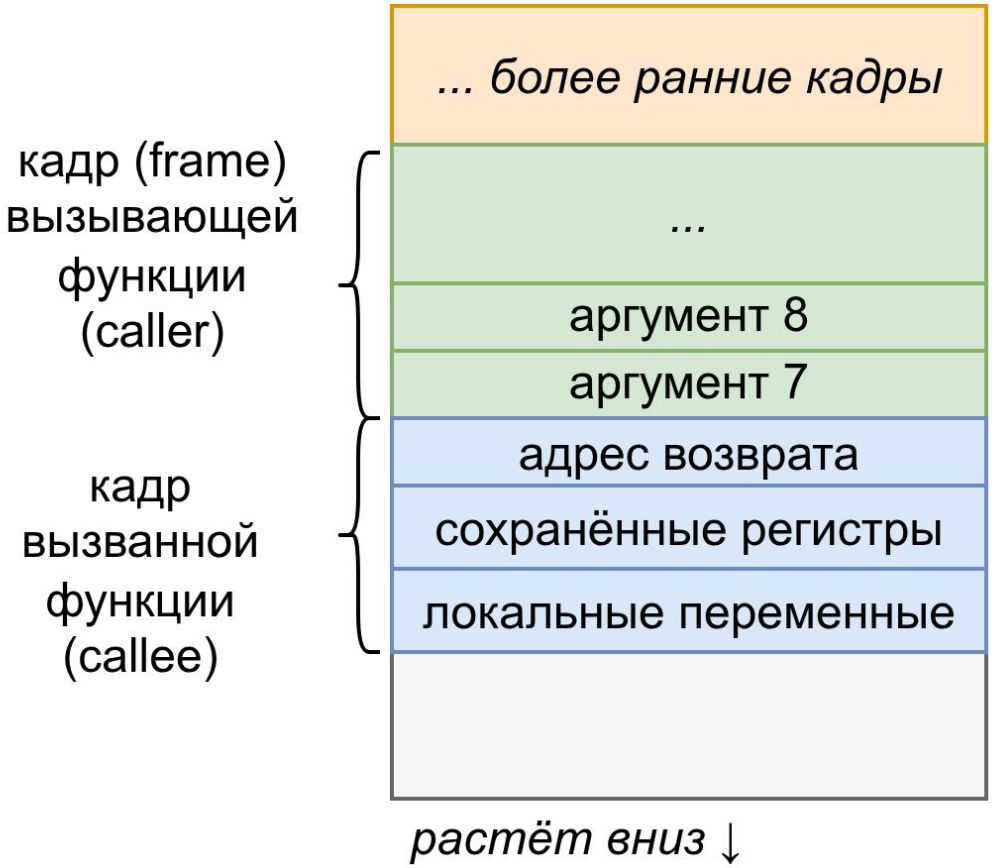
регистры SSE:
128 бит, числа с плавающей точкой

%xmm0
%xmm1
...
%xmm16

Сегменты памяти процесса



Стек в архитектуре x86-64 (64-битные Intel/AMD)



Ссылки

1. Caching and Performance of CPUs
2. Основы виртуальной памяти
3. Регистры процессора
4. Stack frame layout on x86-64
5. Устройство Стека для Intel386
6. What are the calling conventions for UNIX & Linux system calls (and user-space functions) on i386 and x86-64
7. System V Application Binary Interface (PDF)

Конец!