КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 01 — Основы низкоуровневого программирования

Преподаватель: Поденок Леонид Петрович, 505a-5

+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/

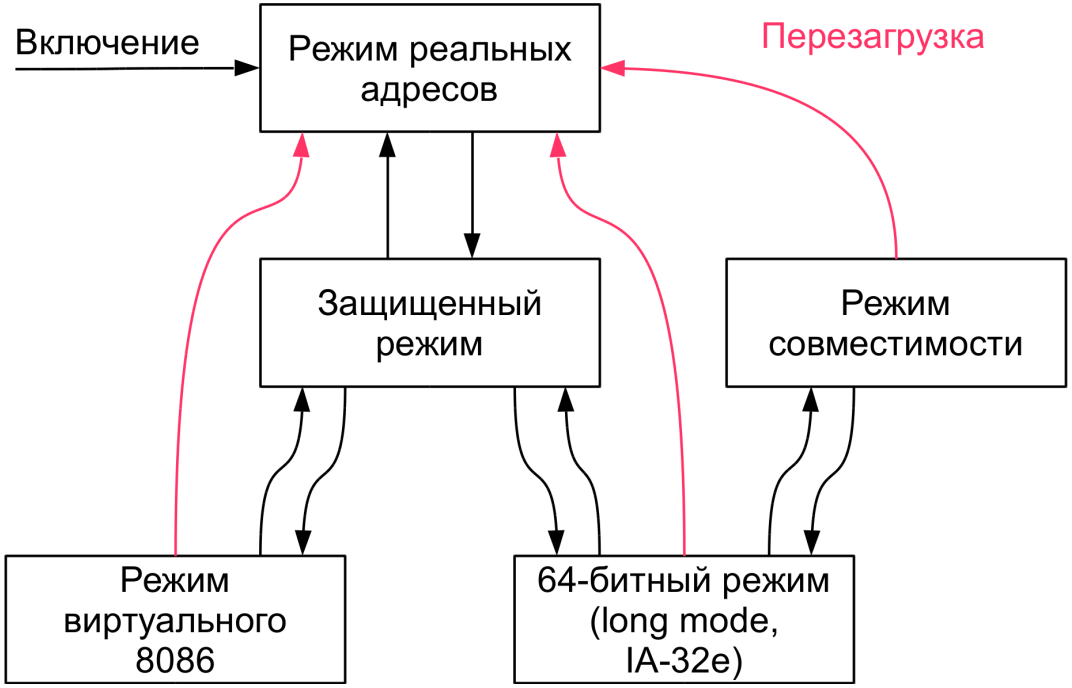
Кафедра ЭВМ, 2022

2022.02.16

Оглавление

Режимы работы процессора AMD-64 (x86-64 или IA-32e)

Режим процессора x86_64	Intel	AMD	Разрядность, бит		Объем
			Адрес	Данные	
реальный			20	16	1М+64к
32 (защищённый)	IA-32	legacy mode	32(36)	32	4(64)Г
виртуальный 8086			16	16	
x64	IA-32e	long mode	64	32(64)	4096(64)Т
совместимости с 32			32(16)	32(16)	
SMM					



Адресация данных

Данные, непосредственно участвующие в обработке, могут находиться:
- в «известном» данной команде месте;
- в команде; // непосредственная адресация
- в регистрах; // регистровая
- в ОЗУ.

```
rep  movsd                ; (esi, edi, ecx)
     mov  eax, 12345678h
     inc  ebx
     mov  eax, [ebx]      ; косв. регистр.
     mov  ecx, cnt
```

1) непосредственная — операнд(ы) в команде
2) регистровая — операнд в регистре
3) прямая — операнд в ОЗУ по адресу в команде
4) косвенная регистровая — операнд в ОЗУ по адресу в регистре
5) дважды косвенная — адрес назначения в ОЗУ по адресу в регистре
6) базовая — адрес назначения формируется как сумма базового регистра и смещения
7) индексная — адрес назначения формируется как сумма индексного регистра $\times$ размер данных, базового регистра и смещения
8) относительная — адрес назначения указывается относительно PC
9) с автоинкрементом/декрементом — содержимое индексного регистра меняется до или после выборки операнда

смещение — offset, displaysment.

Сегментация адресного пространства (сегментные модели памяти)

Адресное пространство памяти логически разбивается на несколько смежных участков (сегментов/секций).

В общем случае сегменты могут перекрываться или не полностью покрывать адресное пространство (взгляд со стороны системы в целом или приложения).

Сегмент характеризуется следующими параметрами:

- базовый адрес (Base Adress);
- предел (Limit);
- параметры доступа (Access).

Адресация с использованием сегментной модели — относительно базового адреса.

В общем случае при обращении по некоторому адресу в адресном пространстве сегмента выполняются проверки на права доступа и пределы.

Память, т.о., имеет двумерную организацию — адрес состоит из двух компонентов:

1) номер сегмента;
2) смещение внутри сегмента.

Характер работы с конкретным сегментом контролируется с помощью атрибутов, таких как права доступа или типы операций, разрешенные с данными, хранящимися в сегменте.

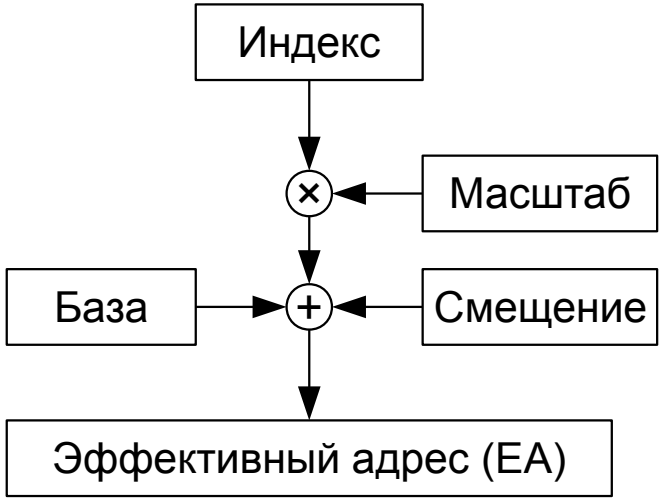
Большинство современных ОС поддерживают сегментную организацию памяти В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием. За разбиение на сегменты отвечают **сегментные регистры**.

Адресация памяти в x86 (IA32)

Существует четыре вида адресов:

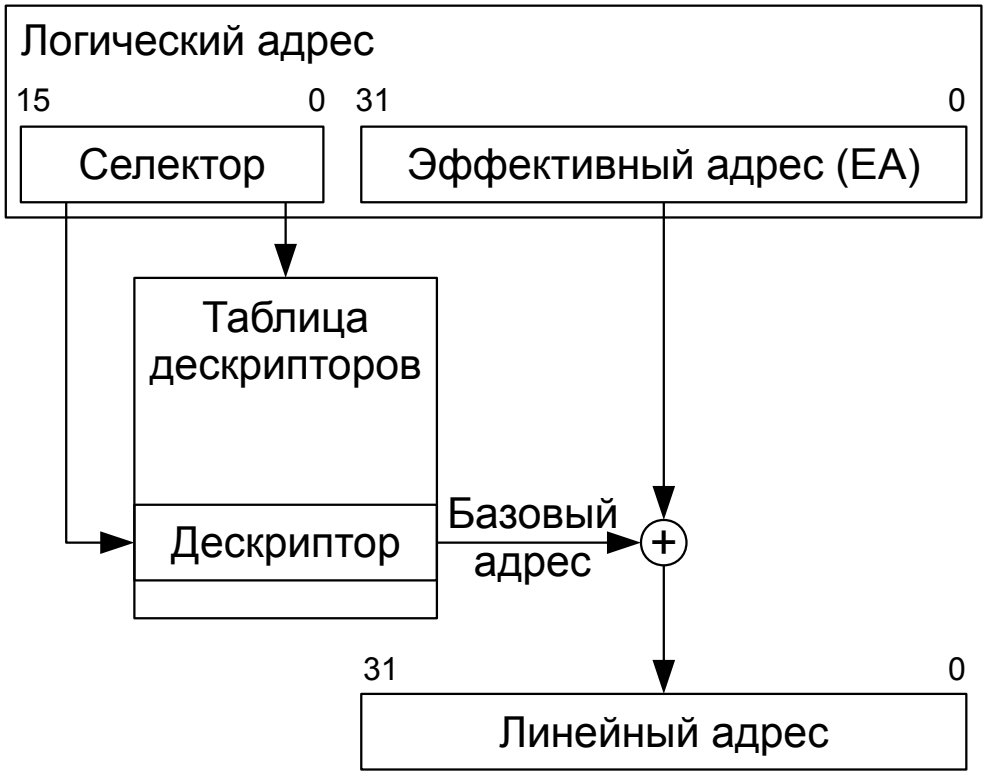
1. эффективный (адрес в сегменте).
2. логический (сегмент и эффективный адрес);
3. линейный (или виртуальный);
4. физический — в системной памяти;

Формирование эффективного адреса

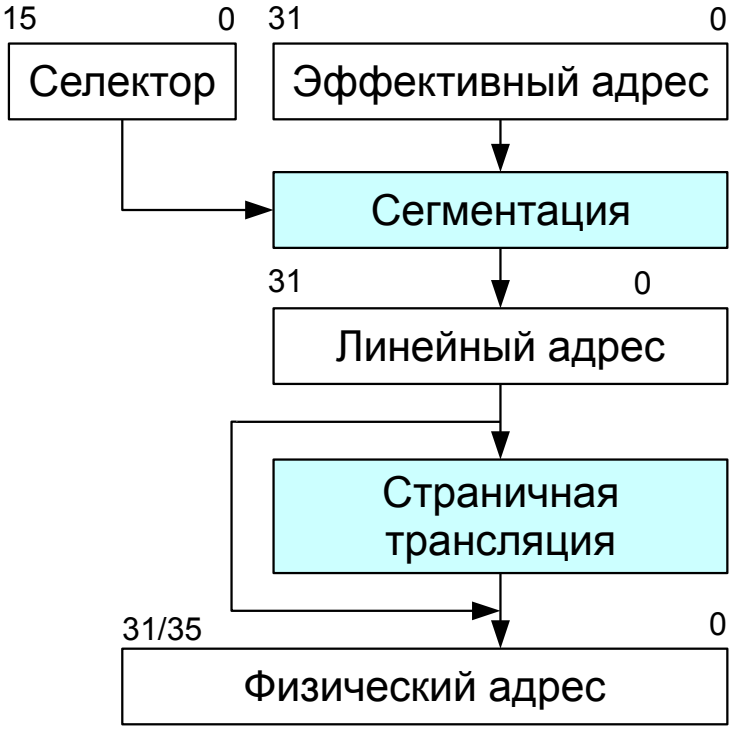


1. из селектора извлекается поле индекса;
2. по индексу находится соответствующий дескриптор;
3. из дескриптора извлекается поле адреса базы.
4. К адресу базы добавляется смещение.

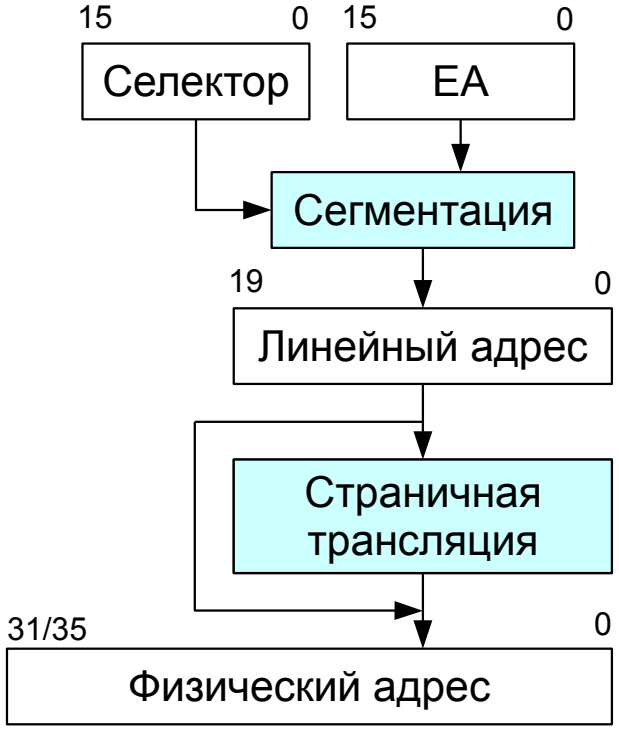
Формирование линейного адреса



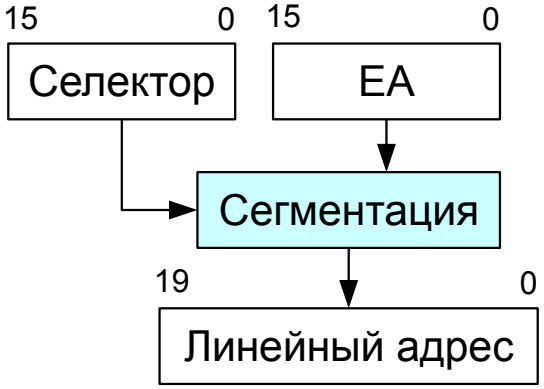
Защищенный режим



V86



Реальный режим



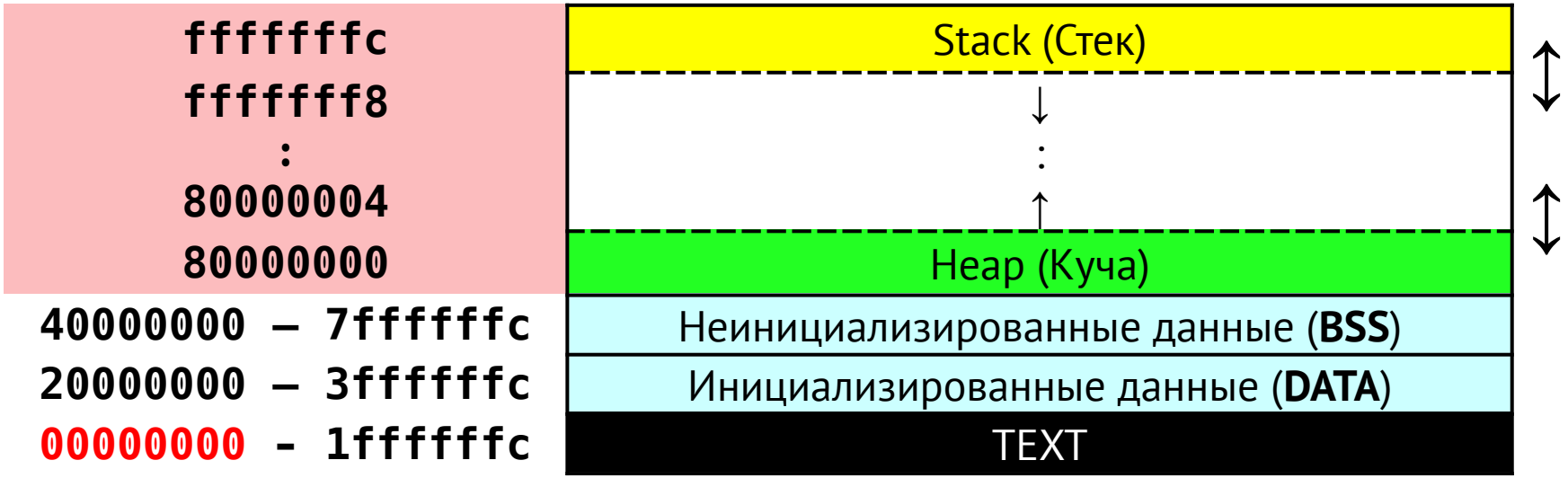
Упрощенная схема организации памяти компьютерной программы

Память компьютерной программы может быть разделена в широком смысле на две части:

- только чтение (**RO**);
- чтение-запись (**RW**).

Ранние ЭВМ держали свою основную программу в постоянной памяти ROM (ПЗУ).

По мере усложнения систем и загрузки программ из других носителей в ОЗУ вместо выполнения их из ПЗУ, сохранялась идея о том, что *некоторые части памяти программы не должны изменяться*. Эти части стали программными сегментами (секциями) **`.text`** и **`.rodata`**, а остальная часть, которая может перезаписываться, разделилась на несколько других сегментов в зависимости от конкретной задачи.



Первая же операция с кучей **`malloc()`** вызывают движение границы секции данных (пунктирный маркер) вверх. Это движение осуществляется системным вызовом **`brk()`**[1], изменяющим расположение *маркера окончания программы* (***program break***), который определяет конец сегмента данных процесса.

Маркер окончания — это первая точка после конца сегмента неинициализированных данных. Увеличение маркера окончания программы позволяет процессу выделить память, уменьшение маркера приводит к освобождению памяти.

Стек начинается с верхней части памяти и растет вниз. Стек не нуждается в явных системных вызовах для увеличения — существует область зарезервированных адресов ниже стека, для которой ядро автоматически выделяет ОЗУ, когда замечает туда попытку записи.

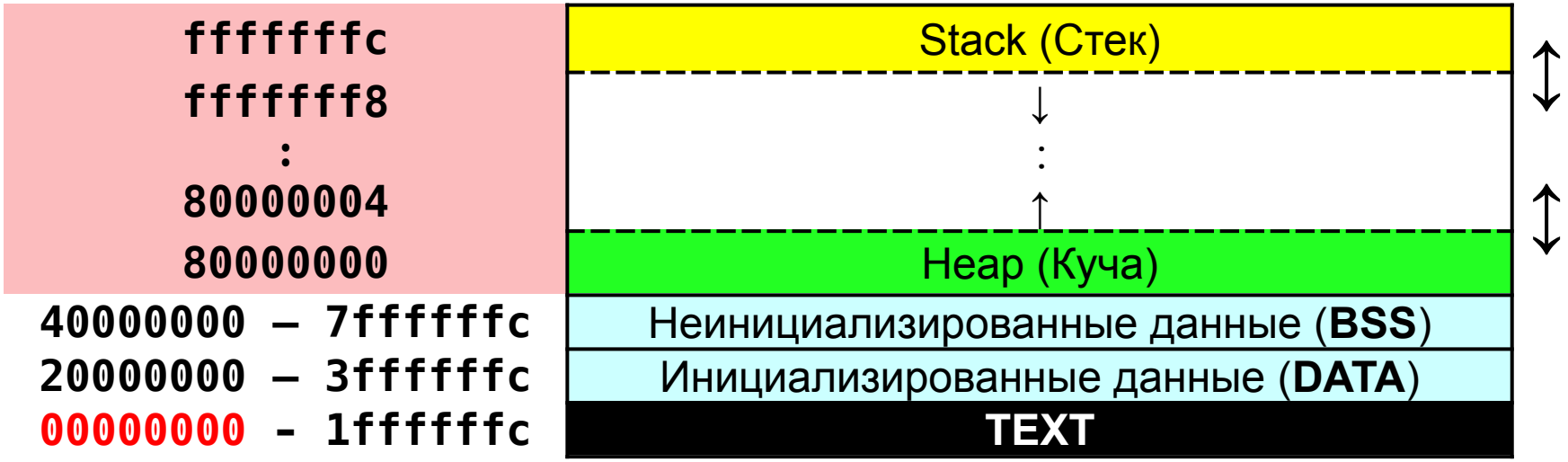
1 brk, sbrk - изменяет размер сегмента данных

Инициализированные данные (Data)

Сегмент **.data** содержит любые глобальные или статические переменные, которые имеют предопределенное значение и могут изменяться. Это любые переменные, которые не определены внутри функций (и, следовательно могут быть доступны из любого места программы) или определены в функции, но определены как статические и сохраняют свое расположение (адрес) при последующих вызовах. Пример на языке С:

```
int  val = 3;
char string[] = "Hello World";
```

Значения этих переменных первоначально сохраняются в постоянной памяти (обычно в **.text**) и копируются в сегмент **.data** во время процедуры запуска программы.

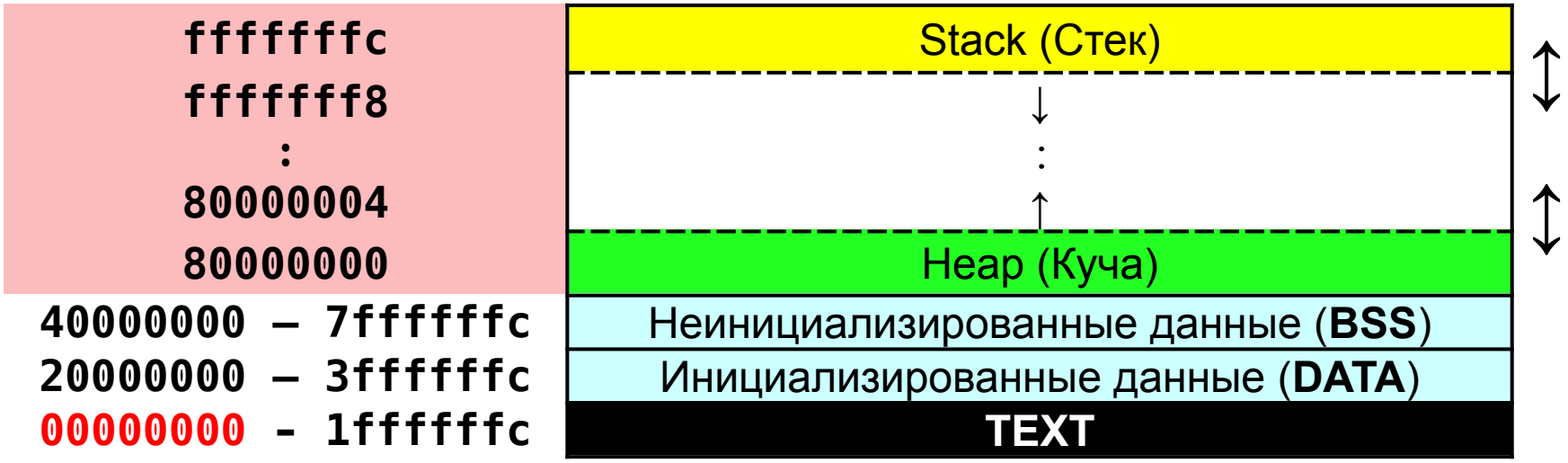


Неинициализированные данные (BSS — Block Started by Symbol)

Сегмент BSS, известный как неинициализированные данные, обычно примыкает к сегменту данных. Сегмент BSS содержит все глобальные переменные и статические переменные, которые инициализируются нулем или не имеют явной инициализации в исходном коде. Например, переменная, определенная как

```
static int i;
```

будет содержаться в сегменте BSS.



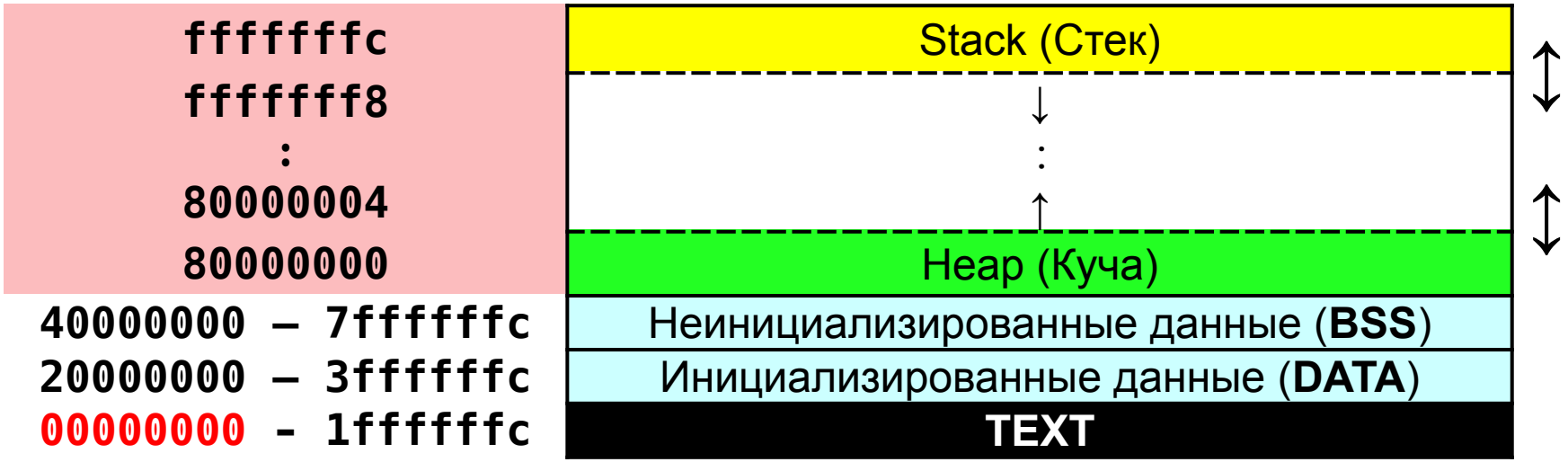
Heap (Куча)

Область кучи обычно начинается в конце сегментов **.bss** и **.data** и растет в сторону увеличения адресов. Область кучи управляется с помощью вызовов

```
malloc()
calloc()
realloc()
free()
```

которые могут использовать системные вызовы **brk()** и **sbrk()** для настройки своего размера.

Системные вызовы **brk()** и **sbrk()** изменяют расположение маркера окончания программы (program break), который определяет конец сегмента данных процесса (т.е., маркер окончания — это первая точка после конца сегмента неинициализированных данных). Увеличение маркера окончания программы позволяет процессу выделить память; уменьшение маркера приводит к освобождению памяти.



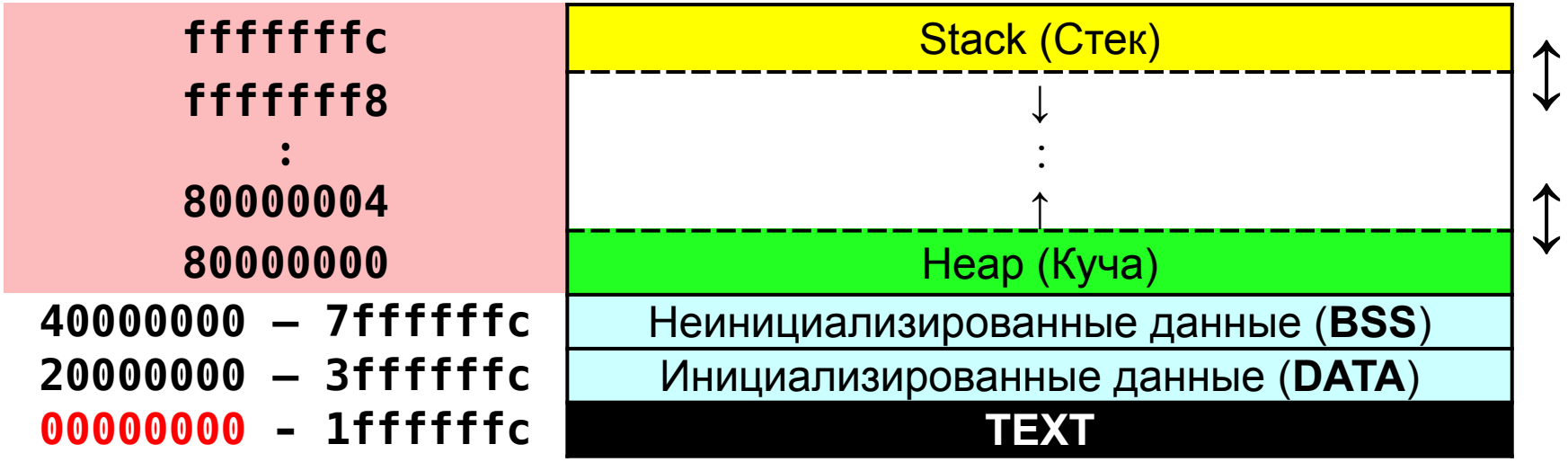
Для реализации вызовов **malloc()/calloc()/realloc()/free()** использование **brk()/sbrk()** и одной «области кучи» не требуется — они могут быть реализованы с использованием системных вызовов **mmap()/munmap()**, которые резервируют потенциально несмежные области виртуальной памяти в виртуальном адресном пространстве процесса.

Область кучи совместно используется всеми потоками, совместно используемыми библиотеками и динамически загружаемыми модулями в рамках процесса.

Stack

Область стека содержит стек программы — структуру LIFO, обычно расположенную в верхних частях памяти. Регистр «указателя стека» отслеживает вершину стека. Она корректируется каждый раз, когда значение «заталкивается» в стек. Набор значений, выделяемых в стеке для одного вызова функции, называется «стековым фреймом». Кадр стека состоит как минимум из адреса возврата. Автоматические переменные также выделяются в стеке.

Область стека всегда традиционно примыкала к области кучи, и они росли навстречу друг другу. Когда указатель стека встречал указатель кучи, свободная память исчерпывалась. При наличии большого адресного пространства и виртуальной памяти они, как правило, размещаются более свободно, но по-прежнему обычно растут в сходящемся направлении. На стандартной архитектуре PC x86 стек растет вниз (по направлению к нулевому адресу), что означает, что более свежие элементы, более глубокие в цепочке вызовов, находятся в более нижних адресах и ближе к куче. На некоторых других архитектурах стек растет в обратном направлении.



Память с точки зрения процесса

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.

Поэтому утверждение о том, что размер двоичного файла не влияет на использование памяти, неверно.

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

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

Чтобы узнать, как распределяется память (во многих системах в любом случае), можно использовать программу на языке C, **`mem_sequence.c`**[2], которая выделяет 5 типов данных, находит их местоположение (виртуальный) адрес памяти, сортирует их по убыванию, а затем отображает.

`mem_sequence.c` работает на Linux, FreeBSD, MacOS X, WinXP и DOS.

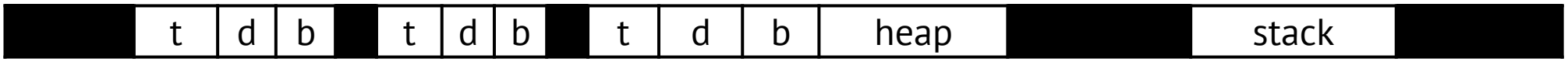
Все UNIX-подобные системы сохраняют похожую модель с небольшими различиями в пороговых значениях адреса, вывод из систем Microsoft отличается.

---

2 См. на ftp://student@lsi.bas-net.by

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 в этом случае выглядит немного сложнее, но суть распределения памяти остается такой же.



0 shlib 1 shlib 2 program 2^{32}

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.

Помимо границы, которую устанавливает **`brk()`**, в «черной» области, могут существовать десятки независимых распределений памяти, сделанные с помощью вызова **`mmap()`** вместо **`brk()`**. Обычно ОС старается располагать их подальше от зоны **`brk()`**, чтобы они не столкнулись.

Жизненный путь программы

Обычно программа проходит нескольких стадий:
- текст на алгоритмическом языке;
- объектный модуль;
- загрузочный модуль;
- бинарный образ в памяти.

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

Виды трансляции:
- компиляция;
- интерпретация;
- динамическая компиляция.

Компилятор (compiler) — транслятор, преобразующий исходный код с какого-либо языка программирования на машинный язык.

Процесс компиляции, как правило, состоит из нескольких этапов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- создание на основе результатов анализов промежуточного кода;
- оптимизация промежуточного кода;
- создание объектного кода, в данном случае машинного.

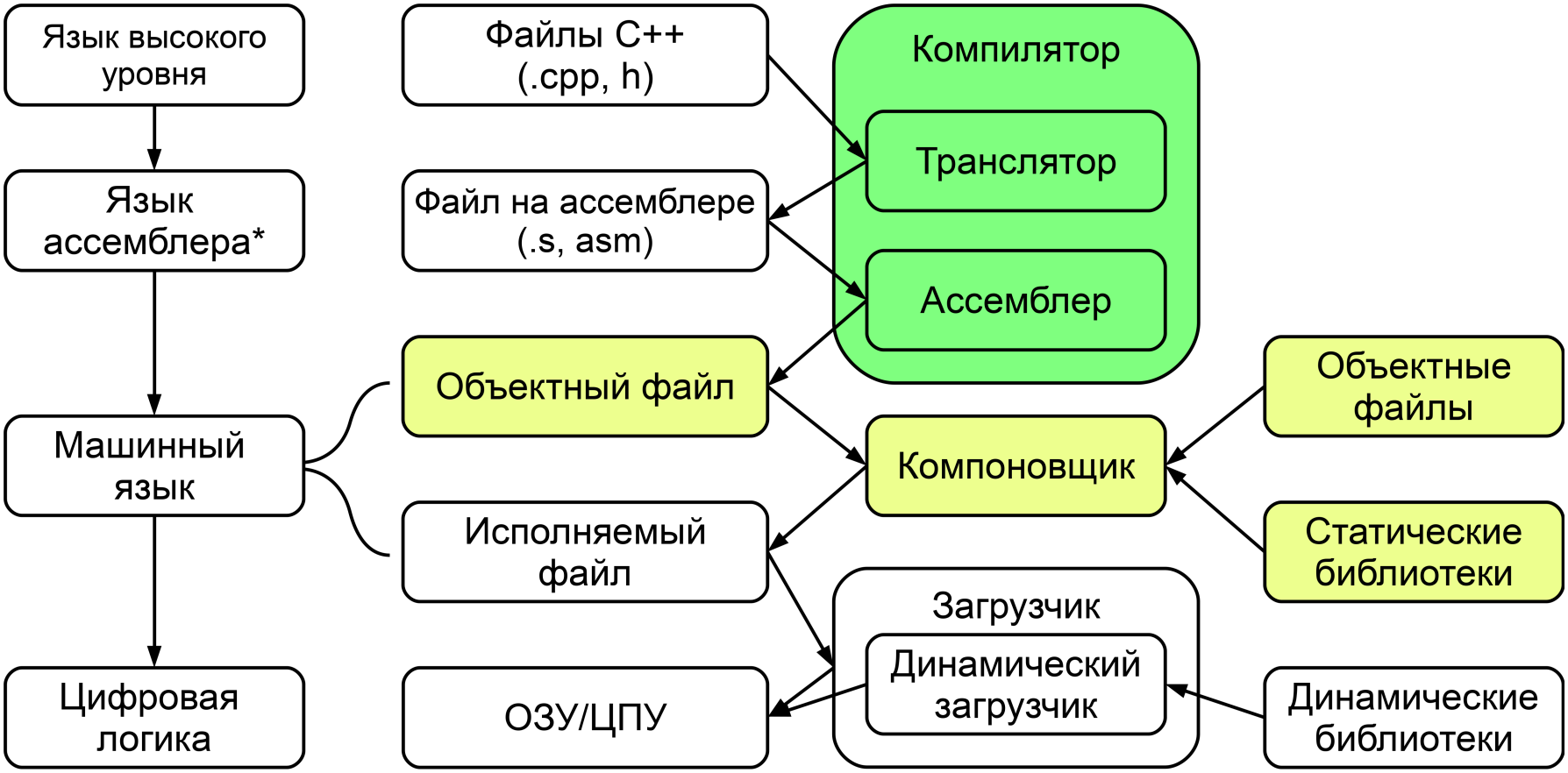
Используемые программой адреса в каждом конкретном случае могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические.

Компилятор связывает эти символические адреса с перемещаемыми адресами (такими как N байт от начала модуля).

Загрузчик или компоновщик (linker), в свою очередь, связывают эти перемещаемые адреса с виртуальными/физическим адресами и создают исполняемый файл.

Каждое связывание — отображение одного адресного пространства в другое.

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).

Привязка инструкций и данных к памяти (настройка адресов) в принципе может быть сделана на следующих шагах:

этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда генерируются абсолютные адреса. Если стартовый адрес программы меняется, необходимо перекомпилировать код. В качестве примера можно привести `.com` программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

этап загрузки (Load time). Если на стадии компиляции не известно где процесс будет размещен в памяти, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одного сегмента памяти в другой, связывание откладывается до времени выполнения. Здесь желательно специализированное оборудование, например регистры перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Например, x86 использует четыре таких (сегментных) регистра.