КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 07 — Память компьютерной программы

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, 2021

2021.10.13

Оглавление

Память компьютерной программы

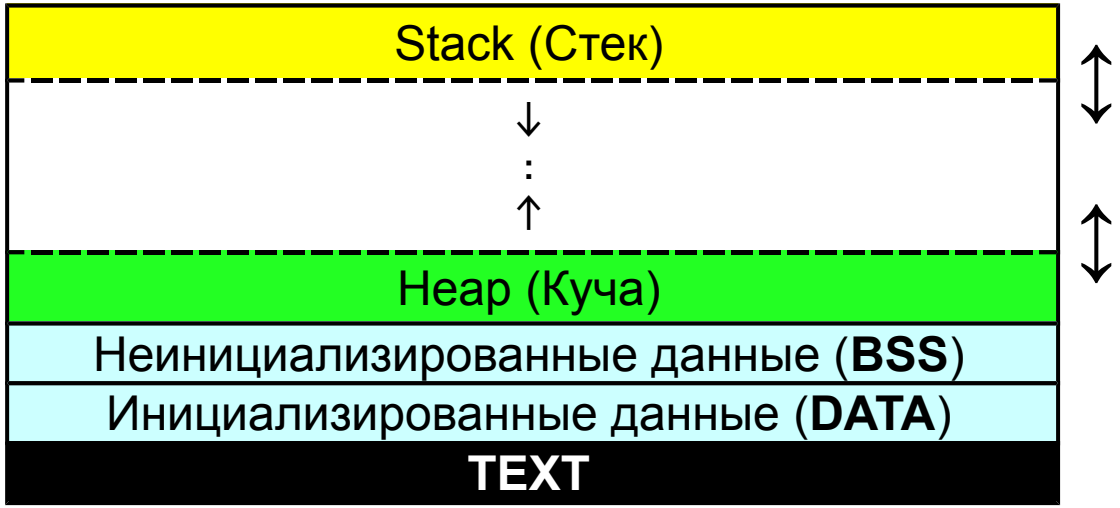
Память компьютерной программы может быть разделена в *широком смысле* на две части:
- только чтение (**RO**);
- чтение-запись (**RW**).

Ранние ЭВМ держали свою основную программу в постоянной памяти, такой как ROM (ПЗУ), PROM (ППЗУ), EPROM (ППППЗУ).

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.

```
ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



Инициализированные данные (Data)

Сегмент **.data** содержит любые глобальные или статические переменные, которые имеют предопределенное значение и могут изменяться.

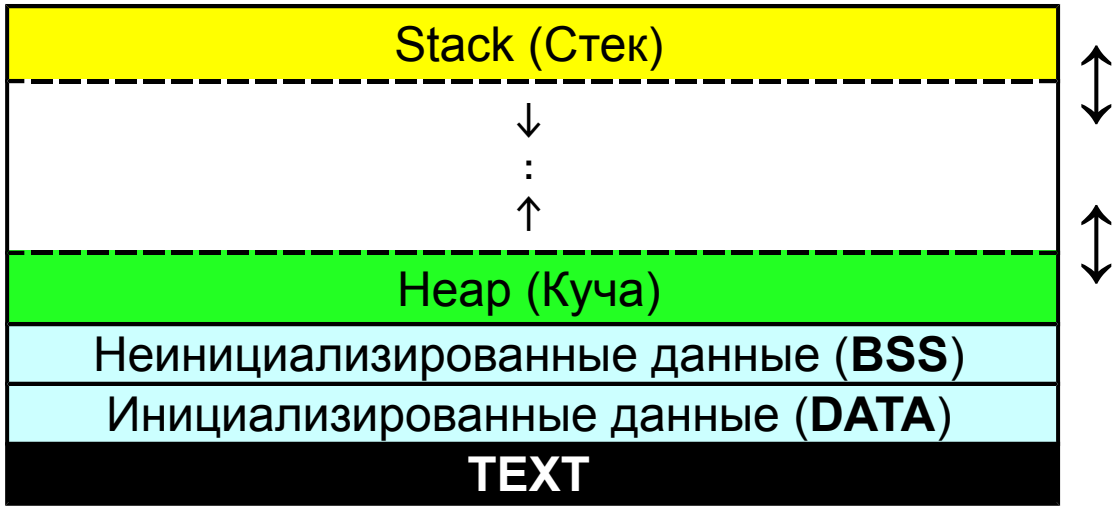
Это любые переменные, которые не определены внутри функций (и, следовательно могут быть доступны из любого места программы) или определены в функции, но определены как статические и сохраняют свое расположение (адрес) при последующих вызовах.

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.

ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc



Неинициализированные данные (BSS — Block Started by Symbol)

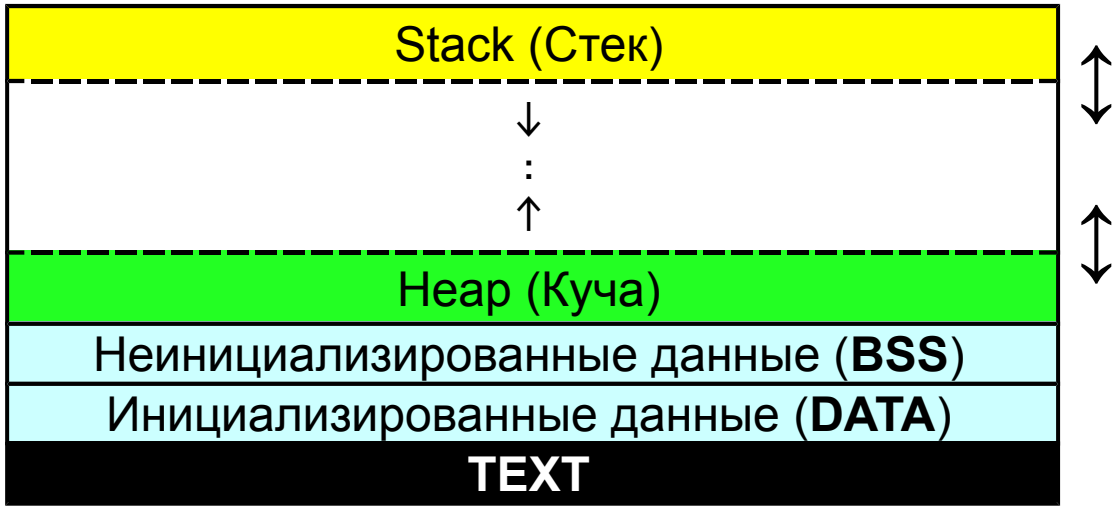
Сегмент BSS, известный как неинициализированные данные, содержит все глобальные переменные и статические переменные, которые инициализируются нулем или не имеют явной инициализации в исходном коде. Обычно примыкает к сегменту данных.

Например, переменная, определенная как

```
static int i;
```

будет содержаться в сегменте BSS.

```
ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



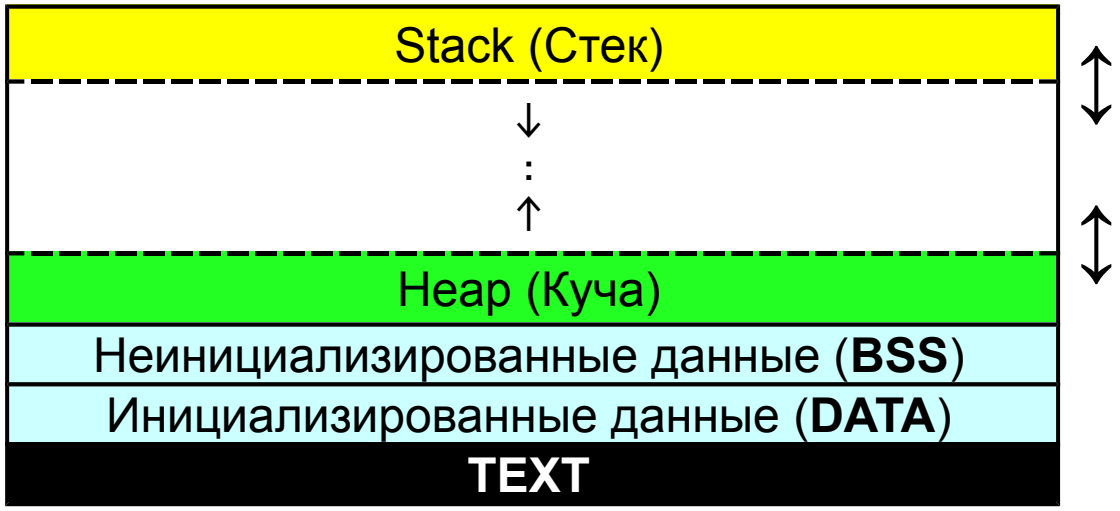
Куча (Heap)

Область кучи обычно начинается в конце сегментов **.bss** и **.data** и растет в сторону увеличения адресов. Область кучи из языка C управляется с помощью функций стандартной библиотеки

```
malloc();
calloc();
realloc();
free().
```

которые используют *системные вызовы* для настройки своего размера.

ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc



Стек (Stack)

Область стека содержит стек программы — структуру LIFO, обычно расположенную в верхних частях памяти.

Вершину стека отслеживает регистр «указателя стека» . Она корректируется каждый раз, когда значение «заталкивается» в стек.

Набор значений, выделяемых в стеке для одного вызова функции, называется «стековым фреймом/кадром».

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 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.

Традиционная организация памяти компьютерной программы

Программный код загружается, начиная с адреса, немного превышающего 0, поскольку указатель со значением **NULL** никуда не указывает.

Секция данных начинается сразу за секцией кода и включает все секции и подсекции с данными — DATA, BSS, и прочие, если присутствуют.

Программа для выполнения считывается в память, где и остается вплоть до своего завершения. Поэтому утверждение о том, что размер вашего двоичного файла не влияет на использование памяти, неверно.

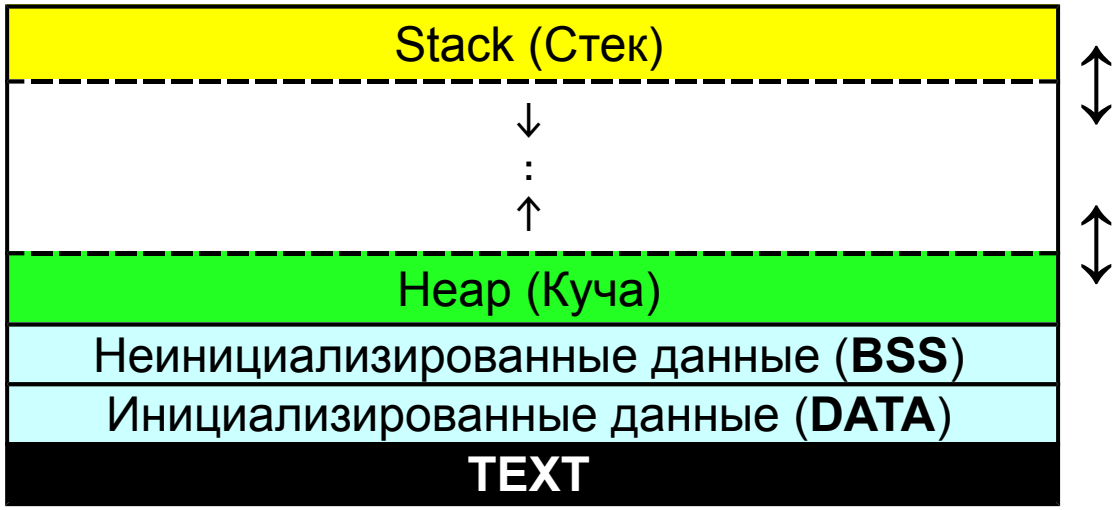
Куча располагается сразу за секцией данных.

При загрузке программы размер кучи нулевой

Стек располагается по верхним адресам. Размер стека устанавливается на этапе загрузки. Стек растет вниз, куча — вверх.

Первая же операция с кучей **malloc()** вызывают движение границы секции данных (пунктирный маркер) вверх.

```
ffffffc
ffffff8
:
80000004
80000000
40000000 – 7ffffffc
20000000 – 3ffffffc
00000000 - 1ffffffc
```



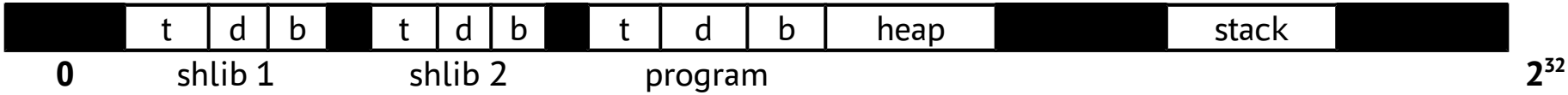
Стек не нуждается в явных системных вызовах для увеличения:

- либо он запускается с выделением для него столько оперативной памяти, сколько он может иметь (традиционный подход);
- либо существует область зарезервированных адресов ниже стека, для которой ядро автоматически выделяет ОЗУ, когда замечает туда попытку записи (это современный подход).

В любом случае, в нижней части адресного пространства, которую можно использовать для стека, может существовать или существовать «защитная» область (**guard area**).

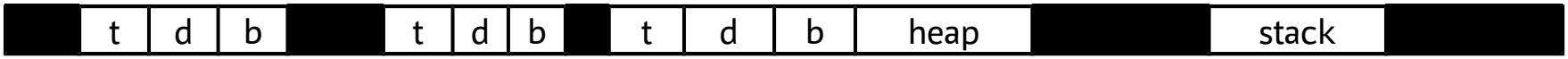
Если такая область существует (все современные системы это делают), она никогда не отображается на физическую память и если либо стек, либо куча пытаются в него «врасти», немедленно возникает исключение ошибки сегментации.

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



Традиционное ядро не устанавливает границ — стек может дорасти до кучи или куча может дорасти до стека, они будут портить данные друг друга, и программа аварийно завершит работу. Если очень повезет, это случится сразу после запуска.

Эту диаграмму *не следует* интерпретировать всеобъемлющим образом — тем, что в точности делает любая конкретная ОС, в том числе и Linux — Linux помещает исполняемый файл гораздо ближе к нулевому адресу, чем совместно используемые библиотеки.



0 shlib 1 shlib 2 program $\mathbf{2^{32}}$

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.

Светлые области — это программа и ее совместно используемые библиотеки (отображаемых в пространство программы таких библиотек могут быть десятки).

У каждой совместно используемой библиотеки есть свои собственные сегменты кода, данных и bss.

Куча не обязательно будет смежной с сегментом данных исполняемой программы, по крайней мере Linux для 64-разрядных программ обычно этого не делает.

Стек не привязан к вершине виртуального адресного пространства, а расстояние между кучей и стеком для 64-разрядных программ настолько велико, что обычно не нужно беспокоиться о его пересечении.

Динамическое выделение памяти в C — <stdlib.h>

malloc, **free**, **calloc**, **realloc** — распределяют и освобождают динамическую память

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
```

malloc()

Функция **malloc()** распределяет **size** байтов и возвращает указатель на распределённую память. Память при этом не инициализируется.

Если значение **size** равно 0, то **malloc()** возвращает или **NULL**, или уникальный указатель, который можно без опасений передавать **free()**.

free()

Функция **free()** освобождает место в памяти, указанное в **ptr**, которое должно быть получено ранее вызовом функции **malloc()**, **calloc()** или **realloc()**.

В противном случае (или если вызов **free(ptr)** уже выполнялся) дальнейшее поведение не определено.

Если значение **ptr** равно **NULL**, то не выполняется никаких действий.

Функция **free()** ничего не возвращает.

realloc()

```
void *realloc(void *ptr, size_t size);
```

Функция **realloc()** меняет размер блока памяти, на который указывает **ptr**, на размер, равный **size** байт.

Содержимое памяти от начала области в пределах наименьшего из старого и нового размеров не будет изменено.

Если новый размер больше старого, то добавленная память не будет инициализирована.

Если **ptr** равно **NULL**, то вызов эквивалентен **malloc(size)** для всех значений **size**.

Если значение **size** равно нулю и **ptr** не равно **NULL**, то вызов эквивалентен **free(ptr)**.

Функция **realloc()** возвращает указатель на новую распределённую память, выровненную должным образом для любого встроенного типа. Возвращаемый указатель может отличаться от **ptr**, или равняться **NULL**, если запрос завершился с ошибкой.

Если значение **size** было равно нулю, то возвращается либо **NULL**, либо указатель, который может быть передан **free()**.

Если **realloc()** завершилась с ошибкой, то начальный блок памяти остаётся нетронутым.

calloc()

```
void *calloc(size_t nmemb, size_t size);
```

Функция **calloc()** распределяет память для массива размером **nmemb** элементов по **size** байтов каждый и возвращает указатель на распределённую память.

Данные в выделенной памяти при этом обнуляются.

Если значение **nmemb** или **size** равно **0**, то **calloc()** возвращает или **NULL**, или уникальный указатель, который можно без опасений передавать **free()**.

Функции **malloc()** и **calloc()** возвращают указатель на распределённую память, выровненную должным образом для любого **встроенного типа**.

При ошибке возвращается **NULL**. Значение **NULL** также может быть получено при успешной работе вызова **malloc()**, если значение **size** равно нулю, или **calloc()** — если значение **nmemb** или **size** равно нулю.

Ошибки

Функции **calloc()**, **malloc()**, **realloc()** могут завершаться со следующей ошибкой:

ENOMEM — не хватает памяти. В этом случае скорее всего приложением достигнут лимит **RLIMIT_AS** или **RLIMIT_DATA**, описанный в **getrlimit(2)**.