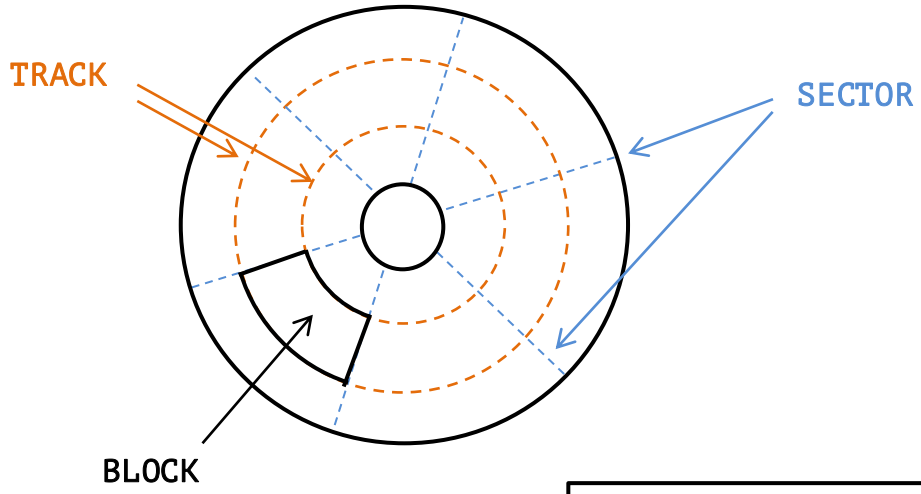
УСТРОЙСТВО ПАМЯТИ ПРОЦЕССА

***** DISK STRUCTURE and HOW DATA is STORED on disk **********



most common block size = 512 bytes ⟶ 0 511

ADRESS = (TRACK, SECTOR) + OFFSET

OFFSET

Data on disk dealing is about DBMS (database management system). Data cant be used directly in the disk: they should be previously loaded to the RAM. In RAM data presented as Data Structures.

НАПИШЕМ ПРОСТУЮ ПРОГРАММУ

```
1:        #include <iostream>
2:        int main( )
3:        {
4:             int i = 42;
5:             std::cout << &i << std::endl;
6:             system(“pause”);
7:        }
```

Адрес &i может быть в зависимости от реализации ОС:
     или ВИРТАУЛЬНЫМ АДРЕСОМ (чаще всего)
     или ФИЗИЧЕСКИМ = реальным АДРЕСОМ в RAM (random access memory)

Будем рассматривать случай с виртуальным адресом
Пример вывода программы: 0x7afe14

ВИРТУАЛЬНЫЙ АДРЕС - что такое

Виртуальный адрес = адрес, по которому размещается переменная во время ассемблирования (формирования объектного кода).

Диапазон доступных адресов в общем виде всегда:
     является непрерывным диапазоном памяти
     от 0x00...00 до 0xFF...FF
а конкретный размер зависит - снова - от особенностей архитектуры и операционной системы. Целый диапазон называется виртуальным адресным пространством (virtual address space = VAS)

Теоретические размеры:

OS	common	from	to
32-bit	4 GiB	0x 00000000	0x FFFFFFFF
64-bit	16 EiB	0x0000000000000000	0xFFFFFFFFFFFFFFFF

* отсюда и далее речь пойдет о 32-bit системах,
  если не уточняется иное

Вариант проверки, действительно ли адрес виртуальный - запустить исполняемый файл на нескольких других компьютерах. Если всякий раз вывод идентичен, следовательно, адрес виртуальный.

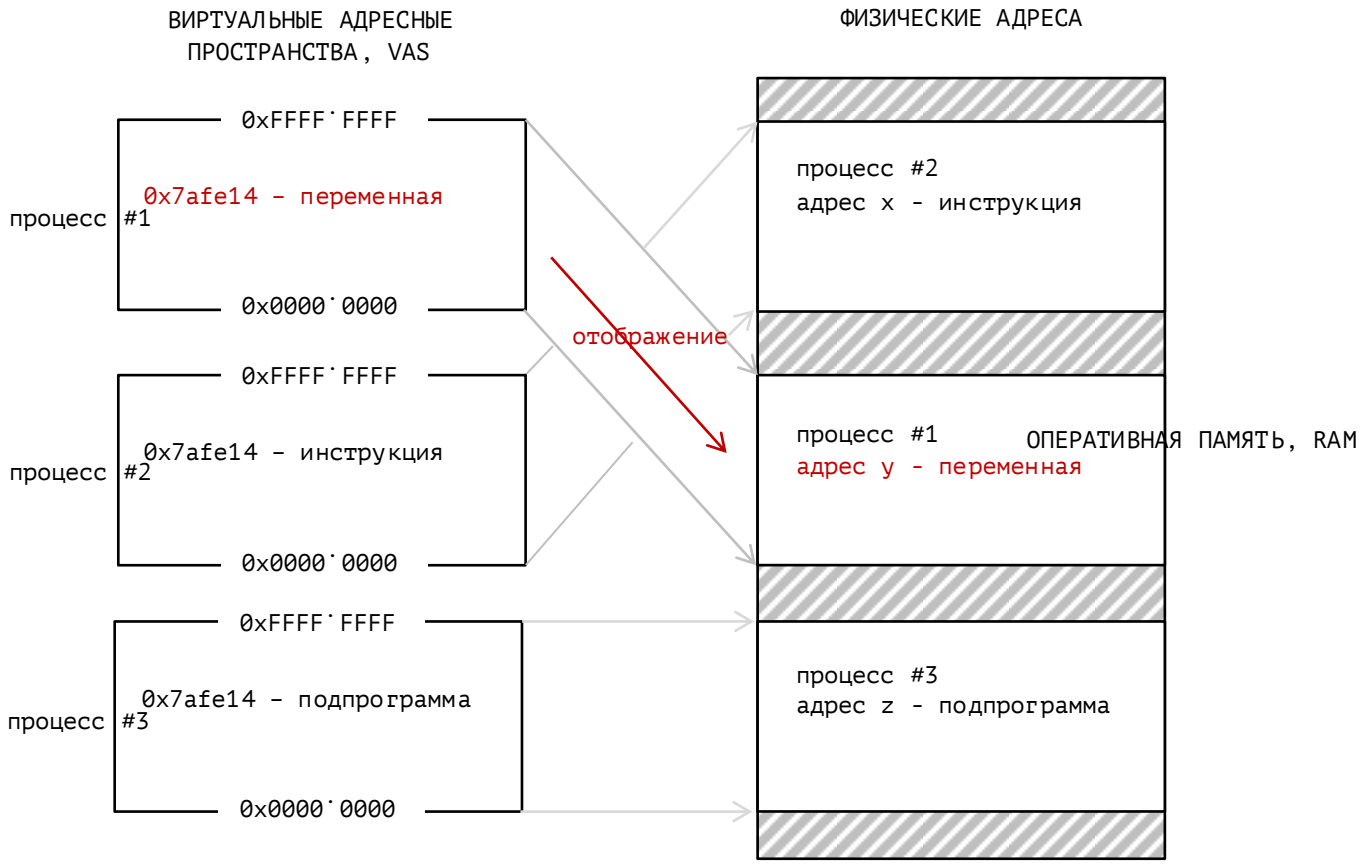
ВИРТУАЛЬНЫЙ АДРЕС – несколько процессов

ПРОЦЕСС (простыми словами) – абстракция, которая обозначает совокупность всех системных ресурсов, связанных с выполнением кода запущенного исполняемого файла. Эта абстракция строится на основе независимого виртуального адресного пространства.

Для каждого из исполняемых файлов на этапе ассемблирования транслятором генерируется собственное VAS с точно таким же диапазоном от 0x0000'0000 до 0xFFFF'FFFF

В таком случае, при запуске обоих процессов одновременно разные значения могут оказаться по идентичным виртуальным адресам. Например, если в вышеприведенной программе в строке 4 заменить значение i на 1, скомпилировать и запустить одновременно с первым вариантом (i = 42), то вывод обоих приложений будет идентичным – значение переменной i из каждой программы хранится по адресу 0x7afe14.

Очевидно, это было бы невозможно, если бы каждая программа имела дело с реальным адресом, по которому хранится 1 или 42. Поэтому общая схема взаимодействия оперативной памяти (будет рассмотрено позже) и процессов выглядит следующим образом:



ЧТО НЕ ТАК С ЭТОЙ СХЕМОЙ?

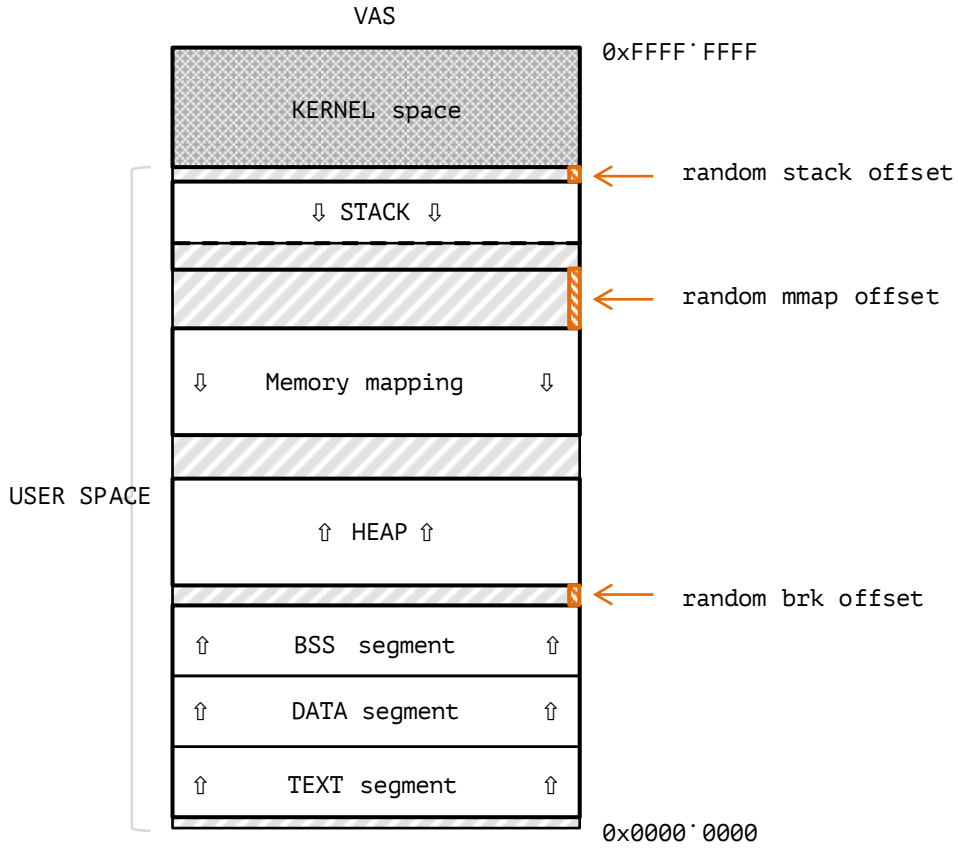
Сопоставление виртуальных адресов физическим называется ОТОБРАЖЕНИЕМ. Внутри себя процесс имеет дело с виртуальным адресами – а с реальными адресами работает аппаратное обеспечение (специальная подсистема RAM).

Как это происходит, будет рассмотрено позднее. Пока что главное:

- изнутри VAS процесс выглядит единственным пользователем RAM
- трансляция адреса из виртуального в реальный – дело RAM

В этой схеме неправильно, что единое VAS отображается в непрерывный блок RAM. На самом деле, значения из VAS могут быть разбросаны по физическим адресам. Сейчас будем более подробно разбираться с устройством VAS конкретного процесса и недостаток будет устранен.

ВИРТУАЛЬНЫЙ АДРЕС – один процесс



KERNEL SPACE и USER SPACE

Пространство VAS, зарезервированное под использование ядром. Обращение по этим адресам недоступно для пользовательского кода, так как оно помечено как привелигированное. Попытка прочитать или записать по этим адресам вызовет ‘Segmentation fault’

Это пространство выделено для ядра, потому что как раз ядро должно обращаться и к собственной памяти, и к памяти процесса. Проще и быстрее всего это сделать если ядро может видеть оба диапазона одновременно.

При этом адресам ‘Kernel space’ каждого выполняемого процесса соответствуют идентичные физические адреса. Другими словами, ‘Kernel space’ каждого процесса являются копиями друг друга.

Соотношение KERNEL SPACE : USER SPACE могут отличаться в зависимости от архитектуры, OS и пользовательских настроек. Например, возможные соотношения:

- для 32-bit = 1:3, 2:2, … GiB
- для 64-bit = 248:8 TB
  - KS = от 0xFFFF·0800·0000·0000 до 0xFFFF·FFFF·FFFF·FFFF
  - US = от 0x0000·0000·0000·0000 до 0x0000·07FF·FFFF·FFFF

TEXT SEGMENT

Адреса, по которым записан:

- сам текст исполняемого кода
- строковые литералы
- бинарный файл

DATA SEGMENT + BSS SEGMENT = **статическая память C++**

Переменные, объявленные:
- в глобальное области видимости
- со статической продолжительность жизни (static)

Разница: Data segment = инициализированные переменные
BSS segment = неинициализированные переменные

Смысл работы следующий:

Когда запускается исполняемый файл, все байты физической памяти, которые ОС выделяет для его отображения из виртуальной памяти, содержат случайные значения, оставшиеся от других файлов. В ‘Text segment’ и ‘Data segment’ эти случайныее значения перезаписываются, так как новые значения напрямую определены пользователем.

А для того, чтобы неинициалзированные глобальные и статические переменные не содержали случайных значений, в момент запуска программы ‘BSS segment’ полностью заполняется нулями. Соответственно все помещенные туда переменные оказываются проинициализированы нулями.

Обратить внимание на то, что даже если значение 0 напрямую задано пользователем: static int i = 0; переменная все равно попадет в ‘BSS segment’

STACK = **автоматическая память C++**

Хранилище данных,организованное по принципу LIFO (last in first out). Реализует последовательное выполнение вызываемых функций и возврат программы - после их завершения – к тому месту, откуда они были вызваны, для продолжения выполнения логики программы.

Достоинство:

- скорость доступа
- не требует дополнительных инструментов очистки

Недостаток:

- туда помещаются только compile time данные
- небольшой объем (в среднем около 8 Мб)

Изначально объем стека может быть выделен меньше максимально доступного (пунктирная линия на иллюстрации). А затем, в случае необходимости, увеличен. Но:

- только до RSTACK_LIMIT (тот самый лимит – нижяя граница)
- после увличения стек не может быть уменьшен

HEAP = **динамическая память C++**

Память, которая выделяется во время выполнения программы
Достоинство:
- большой объем (теоретически ограничен только размером US)
- обработка запроса на выделение - в run time
- позволяет продлить время жизни выделенного ресурса и после завершения работы функции

Недостаток:
- требует дополнительных инструментов очистки
- относительно медленная

MEMORY MAPPING (тоже динамическая)

Сегмент типа HEAP, но отличающийся тем, что его адреса побайтно соответствуют адресам физической памяти. И однажды установленная связь в двльнейшем не меняется – то есть приложение обращается по виртуальным адресам так, как будто это и есть физические адреса.

Приложение может запросить эту память через системный вызов:
- mmap() в Linux
- CreateFileMapping() / MapViewOfFile() в Windows

Как устроено обращение к физическим адресам через виртуальные адреса других сегментов – будет рассмотрено отдельно

RANDOM STACK, MMAP, BRK OFFSET

Случайные смещение начала сегментов для предотвращения хакерских атак с использованием доступа к данным работающего приложения

```
1:    #include <iostream>
2:
3:         int w = 1;
4:         int x = 2;
5:         int y = 0;
6:         int z    ;
7:
8:         void function( )  {
9:              int j = 0;
10:             int k = 1;
11:             std::cout << "  &j =   " << &j << std::endl;
12:             std::cout << "  &k =   " << &k << std::endl;
13:        }
14:
15:        int main( ) {
16:
17:             void* m_ptr = reinterpret_cast<void*>(  main  );
18:             void* f_ptr = reinterpret_cast<void*>(function);
19:             std::cout << "TEXT segment: \n";
20:             std::cout << "  func =  " << f_ptr << std::endl;
21:             std::cout << "  main =  " << m_ptr << std::endl;
22:             std::cout << std::endl;
23:
24:             std::cout << "DATA segment: \n";;
25:             std::cout << "  &w =   " << &w << std::endl;
26:             std::cout << "  &x =   " << &x << std::endl;
27:             std::cout << std::endl;
28:
29:             std::cout << "BSS segment: \n";
30:             std::cout << "  &y =   " << &y << std::endl;
31:             std::cout << "  &z =   " << &z << std::endl;
32:             std::cout << std::endl;
33:
34:             void* ptr1 = malloc(sizeof(int));
35:             void* ptr2 = malloc(sizeof(int));
36:             std::cout << "HEAP: \n";
37:             std::cout << "  ptr1 = " << ptr1 << std::endl;
38:             std::cout << "  ptr2 = " << ptr2 << std::endl;
39:             std::cout << std::endl;
40:
41:             int i = 42;
42:             std::cout << "STACK: \n";
43:             std::cout << "  &i =   " << &i << std::endl;
44:             function( );
45:             std::cout << std::endl;
46:
47:        }
```

Что не так с предыдущей схемой? Проведем эксперимент и несколько раз запустим код, чтобы увидеть, как располагаются рассмотренный сегменты в VAS.

РЕЗУЛЬТАТЫ КОМПИЛЯЦИИ И ЗАПУСКА НА РАЗНЫХ ОС

ОС Windows, x64
GCC 9.2.0
1ый, … запуск

```
TEXT segment:
  func =  0x401560
  main =  0x4015df

DATA segment:
  &u =   0x4b5010
  &v =   0x4b5014
  &w =   0x4b5018

BSS segment:
  &x =   0x4e4030
  &y =   0x4e4034
  &z =   0x4e403c

HEAP:
  ptr1 = 0x021450
  ptr2 = 0x021470

STACK:
  &i =   0x7afdfc
  &j =   0x7afdbc
  &k =   0x7afdb8
```

HEAP ниже всех и растет

ОС Windows, x64
GCC 9.2.0
2ой, … запуск

```
TEXT segment:
  func =  0x401560
  main =  0x4015df

DATA segment:
  &u =   0x4b5010
  &v =   0x4b5014
  &w =   0x4b5018

BSS segment:
  &x =   0x4e4030
  &y =   0x4e4034
  &z =   0x4e403c

HEAP:
  ptr1 = 0xda1450
  ptr2 = 0xda1470

STACK:
  &i =   0x7afdfc
  &j =   0x7afdbc
  &k =   0x7afdb8
```

HEAP выше STACK и растет

ОС Linux Ubuntu


```
TEXT segment:
  func =  0x5580d27ca209
  main =  0x5580d27ca2bd

DATA segment:
  &u =   0x5580d27cd010
  &v =   0x5580d27cd014
  &w =   0x5580d27cd018

BSS segment:
  &x =   0x5580d27cd154
  &y =   0x5580d27cd158
  &z =   0x5580d27cd162

HEAP:
  ptr1 = 0x5580d40f42c0
  ptr2 = 0x5580d40f42e0

STACK:
  &i =   0x7ffd9c3b0054
  &j =   0x7ffd9c3b0030
  &k =   0x7ffd9c3b0034
```

Все на местах, но STACK локально растет

Видим, что описанная схема является наглядным представлением одной из возможных организаций виртуального адресного пространства. Чтобы смотреть конкретную реализацию, нужно пользоваться соответствующим специальным софтом.

СООТВЕТСТВИЕ VAS и RAM

…