Из каких сегментов состоит структура памяти процесса?

Современные операционные системы многозадачны - умеют выполнять несколько процессов одновременно. Любая программа требует некотрое количество памяти для своей работы. Эта память используется для хранения различных состояний нашей программы - например браузеру требуется хранить адреса открытых вкладок… В целях многозадачности - каждой программе выделяется лишь часть памяти компьютера, называемая **виртуальным адресным пространством**.

<> никак не связаны с физическими. Задача преобразования виртуального адреса в физический лежит на операционной системе.

Размер вирутального адреса и размер виртуального адресного пространства зависит от **разрядности** операционной ситемы. **Разрядность ОС** - размер, который занимает указатель(виртуальный адрес). В случае 32 битных ситем указатель занимает всего 4 байта. Что позволяет получать доступ к `2^(32)` байтам. Таким образом - виртуальное адресное пространство в 32битной системе занимает `(2^(32))/(10^9)` = 4.3 гигабайта(но в пишут что 4).

Половину адресного прострасства занимают процессы ОС(0x80000000 до 0xFFFFFFFF). Вторая половина выделяется для нашей программы.

В случае 64битных систем все сложнее. Несложно посчитать, что там адресное пространство занимает несколько террабайт. Но, судя по , ОС ограничивает виртуальное адресное пространство, и выделяет дополнительную память лишь при необходимости.

Как происходит хранение данных в виртуальном адресном пространстве? Для разных задач используются разные сегменты выделенной памяти. Всего сегментов семь:

- **Kernel space** - пространство, выделенное для операционной системы.
  - В случае win32 занимает 2 гигабайта. В случае Linux - 1.
  - Программа не имеет доступа к данному сегменту. Если случайно набрать адрес из этого сегмента возможно получение ошибки.
- **Стек** - сегмент для хранения локальных параметров функций. Там хранятся локальные переменные, адреса возврата.
  - Стек реализован по принципу **LIFO** - last in first out. Этоу концепцую легко представить. Допустим мы печем блины. После того, как тесто достаточно затвердеет, готовый блин кладётся на тарелку. Какой же блин мы съедим первым? Верно - тот, который был испечен в самом конце и сейчас находится на самой верхушке блинной башенки. Похожим образом устроен стек - функция хранит свои данные в некоторой области(блинчике). Если из функции вызывается другая, то создаётся новый блинчик. В новоиспеченном блине содержится **адрес возврата** - адрес по которому нужно вернуть результат работы функции. Первая функция закончит свою работу лишь тогда, когда получит все необходимые данные от других фукнций. Теперь несложно понять, что функция `main` последеней выходит из стека. По своему <> она передаёт ноль, что является индикатором успешного завершения программы.

  - Размер стека достаточно мал(гугл пишет, что 1Мб). Поэтому с ним нужно работать аккуратно. Если вызвать в нем множество функций(или поместить большие данные), получим ошибку `stack overflow` - переполнение стека. В этом несложно убедиться на примере бесконечной рекурсии.
- **Куча** - служит для хранения динамически выделенных данных. Куча имеет достаточно большой размер. Поэтому именно там следует хранить объёмные данные. Скажем нам нужно сохранить волновую форму длительностью 10 секунд, и записанную с битрейтом 96khz. Для этого требуется массив из 960000 элементов типа float. Сохранить такой массив в стеке невозможно, ведь он занимает `96000*10*32/(1.25*(10^7))` примерно 2Мб.
- **BSS/Data** - хранит статически выделенные глобальные переменные. Между этими сегментами есть небольшие различия, но они не так важны при работе.
- **Text** - программа представляет из себя бинарный файл с инструкциями для компьютера. Именно из этого сегмента компьютер получает инструкци к выполнению.

Каким образом связаны указатели и массивы?

Чтобы ответить на вопрос разберёмся с устройством массива как структуры для хранения данных. Массив можно представить как вектор(строку), состоящую из нескольких ячеек. Размер ячейки - размер типа данных, который хранится в массиве.

Как же выделить такую структуру в памяти? Чтобы ответить на этот вопрос ознакомимся с арифметикой указателей. Адрес в памяти - обычное число, записанное в шестнадцатеричной системе счисления. Соответсвенно для адресов определены операции сложения и вычитания.

Создадим указатель, на переменную типа `int32`, и попробуем проинкрементировать данный указатель:

```
#include <iostream>

int main(){
    int32_t alpha = 200;
    int32_t* pointer = &alpha;
    std::cout<<pointer<<std::endl;
    std::cout<<*pointer<<std::endl;

    std::cout<<++pointer<<std::endl;
    std::cout<<*pointer;

    return 0;
}
```

В результате работы программы получим столбец из трех чисел.

0x61ff08 200 0x61ff0c 6422284

Наибольший интерес для нас представляет первое и третье число. Попробуем почситать, что произошло при инкрементировании указателя. Перейдём в удобную для

нас десятичную систему счисления `0x61ff08=6422280`, `0x61ff0c=6422284`. Таким образом, значения нашего указателя увеличелось на 4.

Alt Text

Теперь понятно, что операция инкрементирования прибавляет к адресу данной ячейки размер типа указателя(в нашем случае `int32`).

Аналогичным образом работает операция декрементирования, вычитания числа… Общую <> можно описать так: `pointer+alpha === pointer+sizeof(*pointer)*alpha`(во второй части выражения + - обычное алгебраическое сложение, в первой + - оператор C++).

Именно на этой концепции и строится работа с массивами c++. Когда мы выделяем массив, мы говорим процессору найти для нас некоторое место в памяти. Размер этого места зависит от длины массива и размера данных, которые содеражатся в массиве. Процессор выполняет поставленную перед ним задачу, а мы получаем лишь адрес первого элемента массива. Прибавляя к адресу число `i`, мы получаем адрес `i-ой` ячейки массива.

Для удобства получения данных по данному индексу придумана конструкция `pointer[i]`. Она работает следующим образом: `pointer[i] === *(pointer+i*sizeof(*pointer))`.

Какими способами можно передать данные в функцию

Передача по значению

Чтобы передать по значению используем синтаксис:

```
t1 function(t a)
```

При передаче таким способом в стеке создаётся копия переменной `a`, типа `t`. В этом несложно убедиться, если попробовать изменить значение a внутри функции.

```
void iterA(int a){
    a++;
}

int main(){
    int a = 0;
    iterA(a);
    std::cout<<a;
    return 0;
}
```

После выполнения программы в консоль выведется значение `0`.

Передача по указателю

```
t1 alpha(t* a)
```

При использовании такого синтаксиса мы получаем адрес, по которому хранится переменная a. Используя оператор разименования можно изменять значение по полученному адресу:

```
void iterA(int* a){
    *(a)++;
}

int main(){
    int a = 0;
    iterA(&a);
    std::cout<<a;
    return 0;
}
```

После выполнения программы, получим значение 1 в консоли

Второе преимущество этого способа заключается в том, что нам больше не приходится копировать значение переменной в стек. Представим, что в функции необходимо выполнять операции с большими типами данных(структурой размером n байт).

Если передать такую структуру по значению, то все n байт необходимо <> в стек. При передаче же по указателю, необходимо скопировать лишь 4 - 8байт(в зависимости от ОС). Таким образом, передача по указателю будет осуществляться гораздо быстрее.

Передача по константнтному указателю

```
t1 function(const t* alpha)
```

Различие с предыдущим способом заключается лишь в том, что теперь мы не имеем возможности изменять значение по полученному адресу. Ведь мы указыем не на `t`, а на `const t`.

Такой синтаксис удобен при работе над большими проектами. Второй программист, использующий написанную нами функцию, может быть уверен, что функция не изменит переданные им данные.

Почему низкоуровневая работа с памятью небезопасна

Мы уже поняли, что имеем доступ к виртуальному адресному пространству, выделенному для нашей программы. Если неаккуратно работать с указателями, можем получать ошибки и ожидать от программы некорректного поведения.

Например - изменение данных сегмента kernel space приводит к ошибке segmentation error.