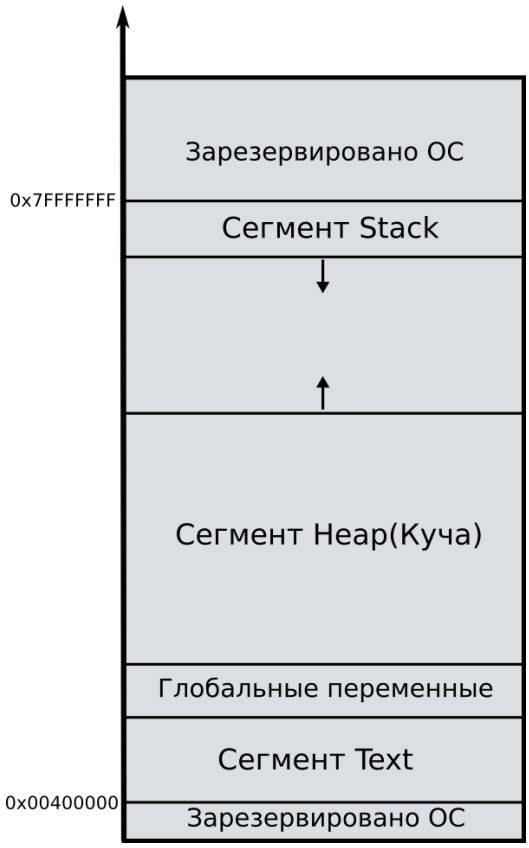
Сегменты памяти.



1. **Сегмент памяти Стек (Stack)**

   - При обычном объявлении переменных и массивов все они создаются в стеке: `float x;` или `int array[10];`
   - Память на эти переменные выделяется в начале функции и освобождается в конце функции.
   - Маленький размер (несколько мегабайт)
   - Быстрее чем куча

2. **Сегмент памяти Куча (Heap)**

   - `malloc` выделяет память в Куче. `int* p = (int*)malloc(10 * sizeof(int));`
   - Память выделяется при вызове `malloc` и освобождается при вызове `free`.
   - Размер ограничен свободной памятью - гигабайты.
   - Медленней чем стек

3. **Сегмент памяти Text**

   - В этом сегменте хранится машинный код программы (код на языке C, сначала, переводится в код на языке Ассемблера, а потом в машинный код).
   - Адрес функции - адрес первого байта инструкций в этом сегменте.

Задачи:

- **Stack overflow:** Переполните стек. (Подсказка: создайте массив на стеке очень большого размера).
- **Размер стека:** Экспериментальным путём найдите размер стека на системе.
- **Переполнение стека через рекурсию:** Переполните стек, используя рекурсию с большой глубиной. (Каждый вложенный вызов функции потребляет память на стеке).

- **Утечка памяти:** В приведённой программе забыли освободить выделенную память. Скомпилируйте и запустите эту программу. Используйте программу `valgrind`, чтобы проверить её на утечки памяти. `valgring ./a.out`

```c
#include <stdlib.h>
int main()
{
        int* p = (char*)malloc(100);
}
```

- **Утечка памяти 2:** Есть ли утечка памяти в следующей программе. Проверить это используя `valgrind`.

```c
#include <stdlib.h>
int main()
{
        char* p = (char*)malloc(100);
        p = (char*)malloc(20);
        free(p);
}
```

- **Valgrind:** Используйте `valgrind`, чтобы проверить на правильность и найти ошибки в программах `valgrind1.c`, `valgrind2.c` и `valgrind3.c`.
- **Двумерный динамический массив:** Создайте в куче двумерный динамический массив из элементов типа `int`. Для этого нужно создать в куче массив элементов типа `int*`, а затем для каждого указателя выделить нужную память.

Связный список

```c
struct node {
    int val;
    struct node* next;
};
typedef struct node Node;

// Вам нужно написать соответствующие функции
// ...
int main()
{
    Node* head = list_create();
    list_add_first(&head, 14);
    for (int i = 0; i < 5; i++)
    {
        list_add_first(&head, i);
        list_add_last(&head, 10 + i);
    }
    printf("%d\n", list_remove_last(&head));
    list_print(head);
    list_reverse(&head);
    list_print(head);
}
```

```c
void list_add_last(Node** p_head, int x)
{
    // Выделяем память на новый элемент
    Node* p_new_node = malloc(sizeof(Node));
    p_new_node->val = x;
    p_new_node->next = NULL;

    // Создаём указатель на первый элемент
    Node* ptr = *p_head;
    if (ptr == NULL)
    {
        *p_head = p_new_node;
    }
    else
    {
        // Идём до последнего элемента
        while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = p_new_node;
    }
}
```

Задачи на связный список:

1. Написать функцию `Node* list_create()`, которая инициализирует список (просто возвращает `NULL`). Примечание: `NULL` - это просто константа равная 0. Её иногда используют вместо нуля для указателей, чтобы различать числовые переменные и указатели.
2. Написать функцию `void list_add_first(Node** p_head, int x)`, которая добавляет элемент x в начало списка. Чтобы добавить элемент, нужно для начала выделить необходимое количество памяти под этот элемент, затем задать поля нового элемента таким образом, чтобы он указывал на начало списка. В конце нужно поменять значение указателя на начало списка. Обратите внимание, что так как нужно изменить значение указателя, то в эту функцию нужно передавать указатель на указатель.
3. Написать функцию `void list_add_last(Node** p_head, int x)`, которая добавляет элемент x в конец списка.
4. Написать функцию `int list_remove_first(Node** p_head)`, которая удаляет элемент из начала списка и возвращает его значение. Не забудьте изменить `*p_head`.
5. Написать функцию `int list_remove_last(Node** p_head)`, которая удаляет элемент из конца списка и возвращает его значение.
6. Написать функцию `void list_print(Node* head)`, которая распечатывает все элементы списка.
7. Написать функцию `int list_size(Node* head)`, которая возвращает количество элементов списка.
8. Написать функцию `int list_is_empty(Node* head)`, которая возвращает 0 если список пуст и 1 если не пуст.
9. Написать функцию `int list_destroy(Node* head)`, которая освобождает всю память, выделенную под список. Так как память выделялась под каждый элемент отдельно, то освобождать нужно также каждый элемент по отдельности.
10. Написать функцию `void list_reverse(Node** p_head)`, которая переворачивает связный список. Первый элемент становится последним, а последний первым.
11. Написать функцию `int list_concatenate(Node** p_head1, Node** p_head2)`, которая добавляет второй связный список в конец первого.
12. Написать функцию `int list_is_loop(Node* p_head)`, которая проверяет, если в связном списке цикл.
13. Реализовать абстрактные типы данных стек(Stack) и очередь(Queue) на основе связного списка.