Учебник по Malloc*

Marwan Burelle
Перевод by bread

1 Введение

Если вы даже не знаете, что это такое, то вам стоит начать с изучения языка Си в среде Unix, прежде чем приступать к чтению этого руководства. Для программиста **malloc** — это функция для выделения блоков памяти в программе на Си. Однако большинство людей не знают, что скрывается за этой функцией. Некоторые даже думают, что это системный вызов или ключевое слово языка. На самом деле, **malloc** — это не более чем простая функция, которую можно понять, имея базовые навыки работы с Си и почти не зная особенностей операционной системы.

Цель этого урока — написать код простой функции **malloc**, чтобы понять основные концепции. Мы не будем создавать эффективную реализацию **malloc**, только базовую. Однако концепция, лежащая в основе, может быть полезна для понимания того, как память управляется в повседневных процессах, а также как работать с выделением, перераспределением и освобождением блоков памяти. С педагогической точки зрения, это хорошая практика для изучения языка Си. Это также полезный материал для понимания того, откуда берутся указатели и как всё организовано в куче.

Что такое malloc?
malloc — это функция стандартной библиотеки языка Си, которая выделяет (то есть резервирует) участки памяти. Она соответствует следующим правилам:

- **malloc** выделяет не менее запрошенного количества байт.
- Указатель, возвращаемый **malloc**, указывает на выделенное пространство (то есть область памяти, в которой программа может успешно читать или записывать данные).
- Никакой другой вызов **malloc** не выделит это пространство или какую-либо его часть, если только указатель не был освобожден ранее.
- **malloc** должен быть эффективным: функция должна завершаться как можно быстрее (она не должна быть NP-трудной!).
- **malloc** также должен поддерживать изменение размера и освобождение памяти.

Функция имеет следующую сигнатуру:

```
void* malloc(size_t size);
```

Где **size** — запрашиваемый размер памяти. Возвращаемый указатель будет равен **NULL** в случае неудачи (например, если не осталось свободной памяти).

2 Куча и системные вызовы brk, sbrk

Прежде чем написать первую реализацию **malloc**, нам нужно понять, как управляется память в большинстве многозадачных систем. В этой части мы будем придерживаться абстрактной точки зрения, поскольку многие детали зависят от конкретной системы и оборудования.

2.1 Память процесса

Каждый процесс имеет собственное виртуальное адресное пространство, которое динамически транслируется в адресное пространство физической памяти с помощью MMU (блока управления памятью) и ядра. Это пространство разделено на несколько частей. Для наших целей важно знать, что существует как минимум:

- область для хранения кода программы,
- стек, где хранятся локальные и изменчивые данные,
- область для постоянных и глобальных переменных,
- неорганизованное пространство для данных программы, называемое **кучей**.

Куча — это непрерывное (в терминах виртуальных адресов) пространство памяти, которое имеет три границы:

- Начальную точку,
- Максимальный предел (управляемый с помощью функций **getrlimit** и **setrlimit** из библиотеки **sys/resource.h**),
- Конечную точку, называемую **прорывом** (break).

Прорыв (break) отмечает конец отображаемого пространства памяти, то есть той части виртуального адресного пространства, которая имеет соответствие в реальной (физической) памяти. На рисунке 1 показана организация памяти.

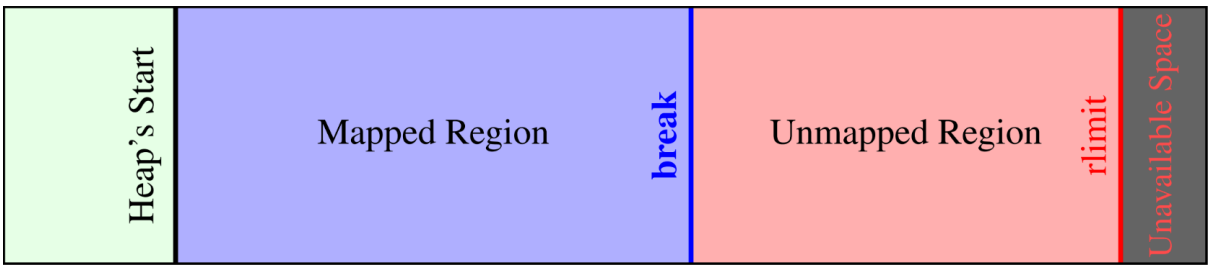

Рисунок 1: Организация памяти кучи

Для реализации **malloc** нам нужно знать, где начинается куча и где находится точка прорыва. Кроме того, нам нужно иметь возможность перемещать эту точку. Для этого предназначены два системных вызова: **brk** и **sbrk**.

2.2 brk и sbrk

Описание этих системных вызовов можно найти на страницах их руководств:

```
int brk(const void *addr);
void* sbrk(intptr_t incr);
```

brk устанавливает точку прорыва (break) по заданному адресу **addr** и возвращает **0** в случае успеха или **-1** в случае ошибки. Глобальная переменная **errno** указывает на характер ошибки.

sbrk перемещает точку прорыва на заданное приращение (в байтах). В зависимости от реализации системы, возвращается либо предыдущий, либо новый адрес прорыва. В случае неудачи возвращается *(void)-1, и устанавливается **errno**. В некоторых системах **sbrk** принимает отрицательные значения, чтобы освободить часть отображаемой памяти.

Поскольку спецификация **sbrk** не определяет точное значение её возвращаемого результата, мы не будем полагаться на это значение при перемещении точки прорыва. Однако мы можем использовать специальный случай **sbrk**: когда приращение равно нулю (т.е. вызов **sbrk(0)**), возвращаемое значение является текущим адресом прорыва (предыдущий и новый адреса прорыва совпадают). Таким образом, **sbrk(0)** можно использовать для получения текущего адреса прорыва, который является началом кучи.

Мы будем использовать **sbrk** в качестве основного инструмента для реализации **malloc**. Всё, что нам нужно сделать, — это запросить дополнительное пространство (если это необходимо) для выполнения запроса на выделение памяти.

2.3 Неотмеченная область и "ничейная земля"

Ранее мы упоминали, что точка прорыва (break) обозначает конец отображаемого виртуального адресного пространства: попытка доступа к адресам, расположенным выше точки прорыва, должна вызывать ошибку (например, ошибку шины). Пространство между точкой прорыва и максимальным пределом кучи не ассоциируется с физической памятью и не управляется менеджером виртуальной памяти системы (MMU и соответствующей частью ядра).

Однако, если вы немного знакомы с концепцией виртуальной памяти (или даже если подумаете несколько секунд), вы знаете, что память отображается постранично: как физическая, так и виртуальная память организованы в страницы (фреймы для физической памяти) фиксированного размера. Чаще всего размер страницы значительно больше одного байта (в большинстве современных систем размер страницы составляет 4096 байт). Таким образом, точка прорыва может не находиться на границе страницы.

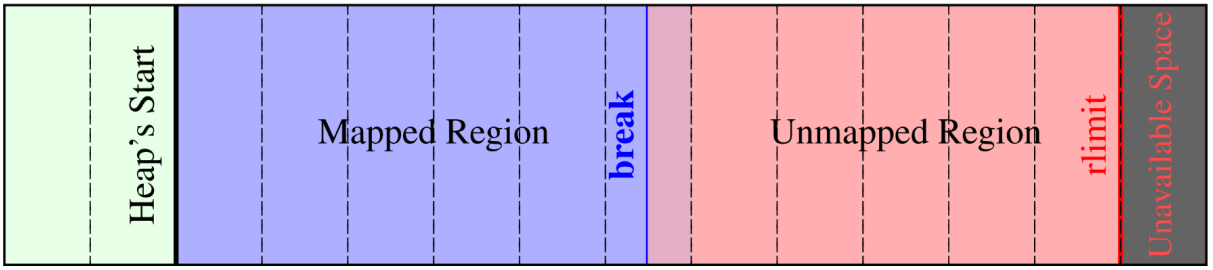

Рисунок 2: Страницы и куча

На рисунке 2 представлена предыдущая организация памяти с учётом границ страниц. Мы видим, что точка прорыва (break) может не совпадать с границей страницы. Каково состояние памяти между точкой прорыва и следующей границей страницы? На самом деле, это пространство доступно! Вы можете получить доступ (для чтения или записи) к байтам в этом пространстве. Проблема в том, что у вас нет никакой подсказки о положении следующей границы страницы. Вы можете вычислить её, но это зависит от системы и не поддаётся универсальным рекомендациям.

Эта "ничейная земля" часто становится источником ошибок: некорректные манипуляции с указателями за пределами кучи могут успешно выполняться в небольших тестах, но приводить к сбоям при работе с большими объемами данных.

2.4 mmap

Хотя мы не будем использовать его в этом уроке, вам стоит обратить внимание на системный вызов **mmap**. Он поддерживает анонимный режим (обычно **mmap** используется для прямого отображения файлов в памяти), который может быть использован для реализации **malloc** (полностью или для некоторых специфических случаев).

mmap в анонимном режиме может выделить определённый объём памяти (с гранулярностью размера страницы), а **munmap** — освободить его. Это часто проще и эффективнее, чем классический **malloc**, основанный на **sbrk**. Многие реализации **malloc** используют **mmap** для больших выделений (более одной страницы). Например, в **malloc** OpenBSD используется только **mmap** с некоторыми особенностями для повышения безопасности (например, предпочтение границ страниц для выделения с промежутками между страницами).

3 Фиктивный malloc

Сначала мы поэкспериментируем с **sbrk**, чтобы написать фиктивный **malloc**. Этот **malloc**, вероятно, самый худший из возможных, даже несмотря на то, что он самый простой и самый быстрый.

3.1 Принцип

Идея очень проста: каждый раз, когда вызывается **malloc**, мы перемещаем точку прорыва (break) на требуемое количество места и возвращаем предыдущий адрес точки прорыва. Это просто и быстро, занимает всего три строки кода. Однако такой подход не позволяет освобождать память, и, конечно, повторное выделение с помощью **realloc** невозможно.

Такой **malloc** расходует много памяти, оставляя устаревшие куски памяти неиспользуемыми. Он приведён здесь исключительно в образовательных целях и для того, чтобы опробовать системный вызов **sbrk**. В образовательных целях мы также добавим в наш **malloc** обработку ошибок.

3.2 Реализация

```
/* Ужасный фиктивный malloc */
#include <sys/types.h>
#include <unistd.h>

void *malloc(size_t size)
{
        void *p; p = sbrk(0);
        /* Если sbrk не работает, мы возвращаем NULL */
        if(sbrk(size) == (void*)-1)
                return NULL;
        return p;
}
```

4 Организация кучи

В разделе 3 на предыдущей странице мы представили первую попытку написать функцию **malloc**, но нам не удалось выполнить все требования. В этом разделе мы попытаемся найти такую организацию кучи, чтобы у нас был эффективный **malloc**, а также **free** и **realloc**.

4.1 Что нам необходимо?

Если рассмотреть проблему вне контекста программирования, можно понять, какая информация нам нужна для решения задачи. Возьмем аналогию: вы владеете полем и разделили его на участки, чтобы сдавать их в аренду. Клиенты просят участки разной длины (вы делите поле, используя только одно измерение), которые, как они ожидают, будут непрерывными. Когда они заканчивают пользоваться участком, то возвращают его вам, чтобы вы могли снова сдать его в аренду.

На одной стороне поля у вас есть дорога с программируемой машиной: вы вводите расстояние между началом поля и пунктом назначения (началом вашего участка). Поэтому нам нужно знать, где начинается каждый участок (это указатель, возвращаемый **malloc**), и когда мы находимся в начале участка, нам нужен адрес следующего участка.

Решение состоит в том, чтобы в начале каждого участка поставить указатель, который содержит адрес следующего участка (а также размер текущего участка, чтобы избежать ненужных вычислений). Теперь, когда клиент хочет получить участок определенного размера, мы берём машину и едем от указателя к указателю. Когда мы находим участок, помеченный как свободный и достаточно большой, мы отдаём его клиенту и убираем отметку со указателя. Если мы добираемся до последнего участка (на его указателе нет адреса следующего участка), мы просто едем в конец участка и добавляем новый указатель.

Теперь мы можем перенести эту идею на память: нам нужна дополнительная информация в начале каждого блока (чанка), указывающая, как минимум, размер блока, адрес следующего блока и то, свободен ли он.

4.2 Как представлять информацию о блоках

Таким образом, нам нужен небольшой блок в начале каждого чанка, содержащий дополнительную информацию, называемую **метаданными**. Этот блок содержит как минимум:

- указатель на следующий чанк,
- флаг для пометки свободных чанков,
- размер данных в чанке.

Разумеется, этот блок информации находится перед указателем, возвращаемым **malloc**.

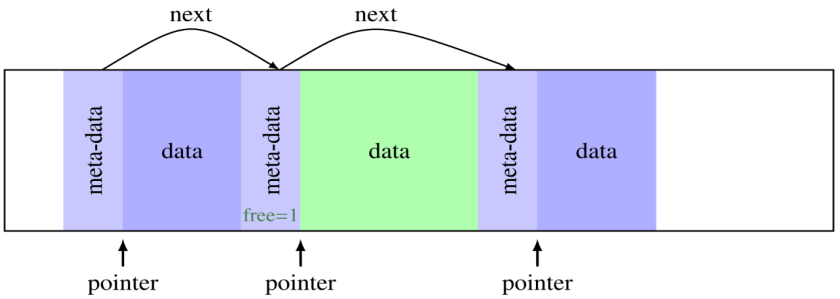

Рисунок 3: Структура фрагментов кучи

На рисунке 3 представлен пример организации кучи с метаданными перед выделенным блоком. Каждый фрагмент данных состоит из блока метаданных, за которым следует блок данных. Указатель, возвращаемый **malloc**, показан в нижней части диаграммы. Обратите внимание, что он указывает на блок данных, а не на весь чанк.

Как перевести это в код на C?

Это выглядит как классический связанный список (и это действительно связанный список), поэтому мы напишем тип для связанного списка с необходимой информацией внутри его элементов. Определение типа является самодокументируемым и не содержит ничего неожиданного:

```
typedef struct s_block *t_block;
struct s_block {
	size_t size;
	t_block next;
	int free;
};
```

Обратите внимание на использование **typedef** для упрощения работы с типом. Поскольку мы никогда не будем использовать тип без указателя, мы включаем указатель в **typedef**. Это хорошая практика для связных списков, так как список — это указатель, а не блок (пустой список — это указатель **NULL**).

Использование **int** в качестве флага может показаться избыточным, но поскольку структура выровнена по умолчанию, это не изменит размер структуры. Позже мы увидим, как можно уменьшить размер метаданных. Ещё один момент, который мы рассмотрим позже, заключается в том, что **malloc** должен возвращать выровненный адрес.

Частый вопрос на этом этапе

Как создать структуру без работающего **malloc**? Ответ прост: нужно только понимать, что такое структура на самом деле. В памяти структура — это просто последовательность её полей. В нашем примере структура **s_block** занимает 12 байт (при 32-битных целых числах): первые 4 байта соответствуют полю **size**, следующие 4 байта — указателю **next**, и последние 4 байта — целому числу **free**.

Когда компилятор встречает обращение к полю структуры (например, **s.free** или **p->free**), он преобразует его в базовый адрес структуры плюс смещение, соответствующее положению поля (так что **p->free** эквивалентно **((char*)p + 8)**, а s.free — *((char*)&s + 8)**).

Всё, что вам нужно сделать, — это выделить с помощью **sbrk** достаточно места для чанка (размер метаданных плюс размер блока данных) и поместить адрес старого прорыва в переменную типа **t_block**:

```
/* Пример использования t_block без malloc */
t_block b;
/* сохраните старое разбиение в b */
b = sbrk(0);
/* добавьте необходимое пространство */
sbrk(sizeof(struct s_block) + size); b->size = size;
```

5 Первый подход к malloc

В этом разделе мы реализуем классический **malloc** с использованием алгоритма **first fit** (первый подходящий). Алгоритм **first fit** очень прост: мы обходим список фрагментов (чанков) и останавливаемся, когда находим свободный блок, в котором достаточно места для запрашиваемого выделения.

5.1 Выровненные указатели

Часто требуется, чтобы указатели были выровнены по размеру целого числа (которое также совпадает с размером указателя). В нашем случае мы будем рассматривать только 32-битный случай. Таким образом, наш указатель должен быть кратен 4 (32 бита = 4 байта). Поскольку наш блок метаданных уже выровнен, нам остаётся только выровнять размер блока данных.

Как это сделать? Есть несколько способов, один из наиболее эффективных — добавить препроцессорный макрос, использующий арифметический трюк.

Арифметический трюк:

Если любое целое положительное число разделить (целочисленное деление) на четыре, а затем умножить на четыре, результат будет ближайшим меньшим кратным четырем. Чтобы получить ближайшее большее кратное четырем, нужно прибавить к числу четыре. Это просто, но этот метод не работает для чисел, уже кратных четырем, так как результат будет следующим кратным.

Давайте рассмотрим арифметику подробнее:
Пусть **x** — целое число, такое, что **x = 4 × p + q**, где **0 ≤ q ≤ 3**.

- Если **x** кратно четырём, то **q = 0**, и **x - 1 = 4 × (p - 1) + 3**.
  Тогда **((x - 1) / 4) × 4 + 4 = 4 × p = x**.
- Если **x** не кратно четырём, то **q ≠ 0**, и **x - 1 = 4 × p + (q - 1)**, где **0 ≤ q - 1 ≤ 2**.
  Тогда **((x - 1) / 4) × 4 + 4 = 4 × p + 4 = (x / 4) × 4 + 4**.

Таким образом, формула **((x - 1) / 4) × 4 + 4** всегда даёт ближайшее большее или равное кратное четырем.

```
#define align4(x) (((((x)-1)>>2)<<2)+4)
```

5.2 Поиск фрагмента: алгоритм First Fit

Найти свободный фрагмент достаточного размера довольно просто:

1. Мы начинаем с базового адреса кучи (который сохраняется в нашем коде, мы увидим это позже).
2. Проверяем текущий фрагмент. Если он подходит, возвращаем его адрес.
3. Если нет, переходим к следующему фрагменту, пока не найдём подходящий или не достигнем конца кучи.

Единственная хитрость — сохранить последний посещенный фрагмент, чтобы функция **malloc** могла легко расширить конец кучи, если подходящий фрагмент не найден.

Код прост: **base** — это глобальный указатель на начальную точку нашей кучи.

```
t_block find_block(t_block *last, size_t size){
        t_block b=base;
        while (b && !(b->free && b->size >= size)){
                *last = b;
                b = b->next;
        }
        return (b);
}
```

Функция возвращает подходящий элемент или NULL, если таковой не найден. После выполнения аргумент last указывает на последний посещенный чанк.

5.3 Расширение кучи

Теперь у нас не всегда будет подходящий фрагмент, и иногда (особенно в начале программы при использовании нашего **malloc**) нам нужно расширить кучу. Это довольно просто: мы перемещаем точку прорыва (break) и инициализируем новый блок. Конечно, нам нужно обновить поле **next** последнего блока в куче.

В дальнейшей разработке нам потребуется выполнить некоторые трюки с размером структуры **struct s_block**, поэтому мы определяем макрос, хранящий размер блока метаданных. Пока что он определяется так:

```
#define BLOCK_SIZE sizeof(struct s_block)
```

Ничего удивительного в этом коде нет. Мы просто возвращаем **NULL**, если **sbrk** не работает (и не пытаемся понять, почему). Заметьте также, что поскольку мы не уверены, что **sbrk** возвращает предыдущий адрес прорыва, мы сначала сохраняем его, а затем перемещаем точку прорыва. Мы могли бы вычислить его, используя **last** и **last->size**.

5.4 Разделение блоков

Вы могли заметить, что мы используем первый доступный блок, независимо от его размера (при условии, что он достаточно большой). Если мы будем так делать, то потеряем много места (представьте: вы запрашиваете 2 байта, а находите блок размером 256 байт). Первое решение — разделение блоков: когда фрагмент достаточно большой, чтобы вместить запрашиваемый размер плюс новый фрагмент (по крайней мере **BLOCK_SIZE + 4**), мы вставляем новый фрагмент в список.

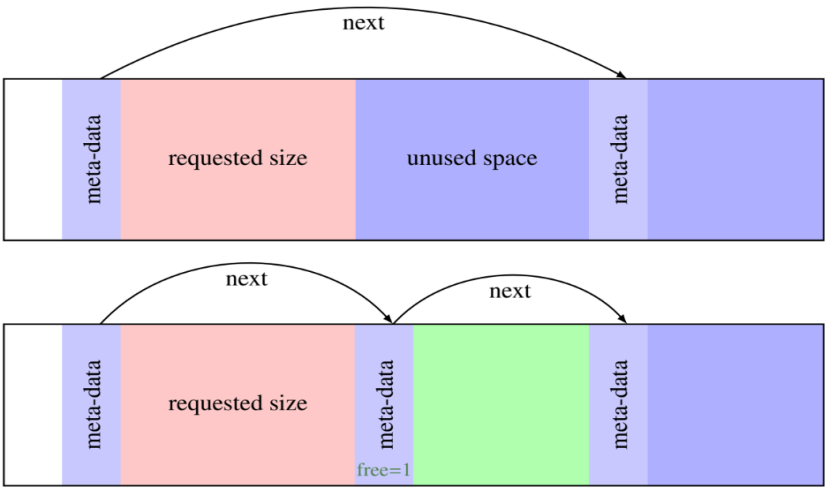

Рисунок 4: Разделение блоков

Следующая функция вызывается только при наличии свободного места. Предоставленный размер также должен быть выровнен. В этой функции нам придется выполнить некоторую арифметику с указателями, чтобы избежать ошибок. Мы используем небольшой трюк, чтобы убедиться, что все наши операции выполняются с точностью до одного байта (помните, что **p + 1** зависит от типа, на который указывает **p**).

Мы просто добавляем в структуру **struct s_block** ещё одно поле типа **char[]** (символьный массив). Массивы в структуре размещаются просто: массив занимает место непосредственно в блоке памяти структуры в той точке, где определено поле. Таким образом, для нас указатель на массив указывает на конец метаданных.

Язык C запрещает массивы нулевой длины, поэтому мы определяем массив длиной в один байт. Это также требует указания макроса для размера структуры **struct s_block**:

```
struct s_block {
        size_t size;
        t_block next;
        int free;
        char data[1];
};
#define BLOCK_SIZE 12 /* 3*4 ... */
```

Обратите внимание, что это расширение не требует модификации функции расширения кучи, так как новое поле **data** не будет использоваться напрямую.

Разделение блока

Эта функция разрезает блок, переданный в аргументе, на блоки данных нужного размера. Как и в предыдущей функции, размер **s** уже выровнен. На рисунке 4 на предыдущей странице показано, что происходит при разделении блока.

```
void split_block (t_block b, size_t s){
        t_block new;
        new = b->data + s;
        new ->size = b->size - s - BLOCK_SIZE ;
        new ->next = b->next;6 new ->free = 1;
        b->size = s;
        b->next = new;
}
```

Обратите внимание на использование **b->data** в арифметике указателей в строке 3. Поскольку поле **data** имеет тип **char[]**, мы уверены, что сложение выполняется с точностью до байта.

5.5 Функция malloc

Теперь мы можем реализовать нашу функцию **malloc**. По сути, она является обёрткой для предыдущих функций. Мы должны выровнять запрашиваемый размер, проверить, вызывается ли **malloc** впервые, и всё остальное уже было описано ранее.

Глобальная переменная base

Во-первых, вспомните, как в разделе 5.2 на странице 8 в функции **find_block** используется глобальная переменная **base**. Эта переменная определяется следующим образом:

```
void *base=NULL;
```

Это указатель типа **void**, и он инициализирован значением **NULL**. Первое, что делает наш **malloc**, — это проверяет значение **base**. Если оно равно **NULL**, значит, это первый вызов **malloc**. В противном случае мы запускаем описанный ранее алгоритм.

Логика работы malloc

Наш **malloc** работает следующим образом:

1. Сначала выравниваем запрашиваемый размер.
2. Если **base** инициализирован (не равен **NULL**):
    - Ищем свободный фрагмент достаточного размера.
    - Если находим подходящий фрагмент:
        - Пытаемся разделить блок (если разница между запрашиваемым размером и размером блока достаточна для хранения метаданных и минимального блока (4 байта)).
        - Помечаем фрагмент как занятый (**b->free = 0**).
    - Если подходящий фрагмент не найден: расширяем кучу.
3. Если **base** не инициализирован (равен **NULL**):
    - Расширяем кучу (которая в этот момент пуста).

Обратите внимание на использование переменной **last** в функции **find_block**: она сохраняет указатель на последний посещённый фрагмент, чтобы мы могли получить к нему доступ во время расширения кучи без повторного обхода всего списка.

Также обратите внимание, что наша функция **extend_heap** работает корректно даже при **last = NULL**.

Каждый раз, когда что-то идёт не так, мы молча возвращаем **NULL**, как и ожидается в спецификации **malloc**. В листинге 1 представлен код.

Листинг 1: Функция malloc

```
void *malloc(size_t size) {
	t_block b, last;
	size_t aligned_size;

	// Выравниваем запрашиваемый размер
	aligned_size = align4(size);

	// Если база инициализирована
	if (base) {
		// Ищем подходящий блок
		last = base;
		b = find_block(&last, aligned_size);

		// Если нашли подходящий блок
		if (b) {
			// Пытаемся разделить блок, если это возможно
			if ((b->size - aligned_size) >= (BLOCK_SIZE + 4)) {
				split_block(b, aligned_size);
			}
			// Помечаем блок как занятый
			b->free = 0;
		} else {
			// Если подходящий блок не найден, расширяем кучу
			 b = extend_heap(last, aligned_size);
			if (!b) {
				return NULL;  // Не удалось расширить кучу
			}
		}
	} else {
		// Если база не инициализирована, расширяем кучу
		 b = extend_heap(NULL, aligned_size);
		if (!b) {
			return NULL;  // Не удалось расширить кучу
		}
		base = b;  // Инициализируем базу
	}

	// Возвращаем указатель на данные
	return b->data;
}
```

6 calloc, realloc и free

6.1 calloc

Функция **calloc** довольно проста:

1. Сначала выполняется **malloc** с правильным размером (произведение двух аргументов).
2. Затем все байты в выделенном блоке обнуляются.

Мы используем небольшой трюк: размер блока данных в фрагменте всегда кратен 4, поэтому итерация выполняется с шагом в 4 байта. Для этого мы просто используем новый указатель как массив беззнаковых целых чисел. Код прост (см. листинг 2).

Листинг 2: Функция calloc

```
void *calloc(size_t number , size_t size ) {
        size_t *new;
        size_t s4 ,i;
        new = malloc(number * size );
        if (new) {
                s4 = align4(number * size) << 2;
                for (i=0; i<s4 ; i++)
                        new[i] = 0;
        }
        return (new );
}
```

6.2 free

Быстрая реализация **free** может быть очень простой, но это не значит, что она эффективна. У нас есть две основные проблемы:

1. Найти фрагмент, который нужно освободить.
2. Избежать фрагментации памяти.

6.2.1 Фрагментация: болезнь malloc

Основной проблемой **malloc** является фрагментация: после нескольких вызовов **malloc** и **free** куча разделяется на множество мелких фрагментов, каждый из которых слишком мал, чтобы удовлетворить большой запрос **malloc**, хотя общее свободное пространство могло бы быть достаточным. Эта проблема известна как **проблема фрагментации памяти**. Хотя мы не можем полностью устранить фрагментацию, вызванную нашим алгоритмом, не изменив его, можно избежать некоторых других источников фрагментации.

Когда мы выбираем свободный фрагмент, достаточно большой для запрашиваемого выделения и ещё одного фрагмента, мы разделяем текущий фрагмент. Хотя это позволяет лучше использовать память (новый фрагмент остается свободным для дальнейшего выделения), это также увеличивает фрагментацию.

Решение для устранения фрагментации — объединение свободных фрагментов. Когда мы освобождаем фрагмент, и его соседи также свободны, мы можем объединить их в один больший фрагмент. Всё, что нам нужно сделать, — это проверить следующий и предыдущий фрагменты. Но как найти предыдущий фрагмент? У нас есть несколько решений:

1. Поиск с самого начала кучи, что особенно медленно (особенно если мы уже выполнили поиск для освобождаемого фрагмента).
2. Если мы уже выполняем поиск для текущего фрагмента, мы можем сохранить указатель на последний посещенный фрагмент (как в функции **find_block**).
3. Сделать наш список двусвязным.

Мы выбираем последнее решение, так как оно проще и позволяет нам выполнять некоторые трюки для поиска целевого фрагмента. Итак, мы снова модифицируем нашу структуру **struct s_block**, но так как у нас есть ещё одна ожидаемая модификация (см. следующий раздел), мы представим ее позже.

Объединение фрагментов

Теперь всё, что нам осталось сделать, — это реализовать слияние. Сначала мы напишем простую функцию слияния, которая объединит текущий фрагмент и его следующий фрагмент. Слияние с предыдущим фрагментом будет просто проверкой и вызовом функции слияния для предыдущего фрагмента. В коде мы используем новое поле **prev** для указания на предыдущий фрагмент.

```
t_block fusion(t_block b){
        if (b->next && b->next ->free ){
        b->size += BLOCK_SIZE + b->next ->size;
        b->next = b->next ->next;
        if (b->next)
                b->next ->prev = b;
        }
        return (b);
}
```

Слияние происходит следующим образом: если следующий фрагмент свободен, мы суммируем размеры текущего и следующего фрагментов, а также размер метаданных. Затем мы обновляем указатель на следующий фрагмент, чтобы

он указывал на фрагмент, следующий за следующим. Если такой фрагмент существует, мы обновляем его указатель на предыдущий фрагмент.

6.2.2 Поиск нужного элемента

Другая проблема **free** заключается в том, чтобы эффективно найти нужный фрагмент, имея только указатель, возвращаемый **malloc**. На самом деле, здесь есть несколько задач:

1. Проверка входного указателя (действительно ли это указатель, выделенный с помощью **malloc**?).
2. Поиск указателя на метаданные.

Большинство недействительных указателей можно исключить с помощью быстрого теста на диапазон: если указатель находится за пределами кучи, он не может быть действительным. Остальные случаи связаны с последней задачей. Как мы можем быть уверены, что указатель был получен с помощью **malloc**?

Решение заключается в том, чтобы добавить в структуру блока некоторое "магическое число". Но лучше, чем магическое число, мы можем использовать сам указатель. Объясняю: допустим, у нас есть поле **ptr**, указывающее на поле **data**. Если **b->ptr == b->data**, то **b**, скорее всего (очень вероятно), является правильным блоком. Вот расширенная структура и функции, которые проверяют и получают доступ к блоку, соответствующему данному указателю:

```
/* структура блока */
struct s_block {
        size_t size;
        struct s_block *next;
        struct s_block *prev;
        int free;
        void *ptr;
        /* Указатель на выделенный блок */
        char data [1];
};
typedef struct s_block *t_block;
/* Получить блок и addr */
t_block get_block (void *p)
{
        char *tmp;
        tmp = p;
        return (p = tmp -= BLOCK_SIZE);
}
/* проверка addr для освобождения */
int valid_addr (void *p)
{
        if (base) {
                if ( p>base && p<sbrk (0)) {
                        return (p == ( get_block (p))->ptr );
                }
        }
        return (0);
}
```

6.2.3 Функция free

Теперь освобождение происходит просто: мы проверяем указатель и получаем соответствующий фрагмент, помечаем его как свободный и, если возможно, объединить с соседними свободными фрагментами. Мы также пытаемся освободить память, если находимся в конце кучи.

Освобождение памяти довольно простое: если мы находимся в конце кучи, нам нужно просто переместить точку прорыва (break) на позицию фрагмента с помощью простого вызова **brk**.

В листинге 3 на следующей странице представлена наша реализация. Она следует следующей структуре:

1. Если указатель действителен:
    - Получаем адрес блока.
    - Помечаем его как свободный.
    - Если предыдущий блок существует и свободен, делаем шаг назад в списке блоков и объединяем два блока.
    - Также пытаемся объединить со следующим блоком.
    - Если мы последний блок, освобождаем память.
    - Если блоков больше нет, возвращаемся в исходное состояние (устанавливаем **base** в **NULL**).
2. Если указатель недействителен, мы молча ничего не делаем.

6.3 Изменение размера фрагментов с помощью realloc

Функция **realloc** почти так же проста, как и **calloc**. По сути, нам нужна только операция копирования памяти. Мы не будем использовать стандартную функцию **memcpy**, так как можем сделать это лучше (размер доступен блоками и выровнен).

Копия проста:

Листинг 3: Функция free

```
void free(void *p)
{
        t_block b;
        if ( valid_addr (p)) {
                b = get_block (p);
                b->free = 1;
                /* слияние с предыдущими, если возможно */
                if(b->prev && b->prev ->free)
                        b = fusion(b->prev );
                /* затем слияние со следующим */
                if (b->next)
                        fusion(b);
                else {
                /* освободим конец кучи */
                        if (b->prev) b->prev ->next = NULL;
                        else base = NULL;
                        brk(b);
                }
        }
}
```

```
/* Копирование данных из блока в блок */
void copy_block (t_block src , t_block dst)
{
        int *sdata ,* ddata;
        size_t i;
        sdata = src ->ptr;
        ddata = dst ->ptr;
        for (i=0; i*4<src ->size && i*4<dst ->size; i++)
                ddata[i] = sdata[i];
}
```

Очень наивная (но рабочая) реализация **realloc** просто следует этому алгоритму:

1. Выделите новый блок заданного размера с помощью **malloc**.
2. Скопируйте данные из старого блока в новый.
3. Освободите старый блок.
4. Верните новый указатель.

Конечно, мы хотим чего-то более эффективного: нам не нужно новое выделение, если текущий блок уже достаточно велик. Различные случаи таковы:

- Если размер не меняется, или дополнительный доступный размер (из-за ограничений выравнивания или если предыдущий размер был слишком мал для разделения) достаточен, мы ничего не делаем.
- Если мы уменьшаем блок, мы пытаемся разделить его.
- Если следующий блок свободен и предоставляет достаточно места, мы объединяем их и, если необходимо, пытаемся разделить.

В листинге 4 на следующей странице представлена наша реализация. Не забывайте, что вызов **realloc(NULL, s)** является допустимым и должен быть заменён на **malloc(s)**.

Листинг 4: Функция realloc

```
void *realloc(void *p, size_t size)
{
        size_t s;
        t_block b, new;
        void *newp;
        if (!p) return (malloc(size));
        if ( valid_addr (p)) {
                s = align4(size);
                b = get_block (p);
                if (b->size >= s) {
                        if (b->size - s >= ( BLOCK_SIZE + 4))
                                split_block (b,s);
                }
                else {
                        /* Попробуем соединить со следующим, если это возможно */
                        if (b->next && b->next ->free
                                && (b->size + BLOCK_SIZE + b->next ->size) >= s)
                        {
                                fusion(b);
```

```
                        if (b->size - s >= ( BLOCK_SIZE + 4))
                                split_block (b,s);
                }
                else
                {
                        /* Старый добрый realloc с новым блоком */
                        newp = malloc(s);
                        /* Мы обречены! */
                        if (!newp) return (NULL );
                        /* Я предполагаю, что это работает! */
                        new = get_block (newp);
                        /* Копирование данных */
                        copy_block (b,new );
                        /* Освободим старый */
                        free(p);
                        return (newp);
                }
        }
        return (p);
    }
    return (NULL );
}
```

6.3.1 FreeBSD's reallocf

FreeBSD предоставляет еще одну функцию **realloc**: **reallocf**, которая освобождает заданный указатель в любом случае (даже если перераспределение не удалось). Это просто вызов **realloc** и освобождение памяти, если мы получаем указатель **NULL**. В листинге 5 представлен код.

Листинг 5: Функция reallocf

```
void *reallocf(void *p, size_t size)
{
        void *newp;
        newp = realloc(p,size );
        if (!newp) free(p);
        return (newp );
}
```

6.4 Собираем все вместе

Теперь всё, что нужно, — это интегрировать модификации, внесенные в структуру блока, в предыдущий код. Нам нужно только переписать функцию **split_block**, функцию **extend_heap** и переопределить макрос **BLOCK_SIZE**.

Листинг 6: Собираем все вместе

```
/* структура блока */
struct s_block {
        size_t size;
        struct s_block *next;
        struct s_block *prev;
        int free;
        void *ptr;
        char data [1]; /* Указатель на выделенный блок */
};
typedef struct s_block *t_block;
/* Определим размер блока, поскольку sizeof будет неверным. */
#define BLOCK_SIZE 20
/* Разделим блок по размеру. */
void split_block (t_block b, size_t s)
{
        t_block new;
        new = (t_block )(b->data + s);
        new ->size = b->size - s - BLOCK_SIZE ;
        new ->next = b->next;
        new ->prev = b;
        new ->free = 1;
        new ->ptr = new ->data;
        b->size = s;
        b->next = new;
        if (new ->next) new ->next ->prev = new;
}
/* Добавляем новый блок в кучу */
/* возвращает NULL, если что-то пошло не так */
t_block extend_heap (t_block last , size_t s)
{
        int sb;
        t_block b;
        b = sbrk (0);
        sb = (int)sbrk( BLOCK_SIZE + s);
        if (sb < 0) return (NULL );
        b->size = s;
        b->next = NULL;
        b->prev = last;
        b->ptr = b->data;
        if (last) last ->next = b;
        b->free = 0;
        return (b);
}
```

Список рисунков

Листинги