**ОБЩАЯ ТЕОРИЯ**

**LAB 4.3**

**Что такое куча (Heap)?**

**Куча** в программировании — это область памяти, которая используется для динамического выделения памяти во время выполнения программы. Это значит, что программа может запрашивать память (например, для хранения данных) в любой момент, а потом освобождать её, когда она больше не нужна.

**Как работает куча?**

Программа запрашивает память (например, через malloc в языке C).

Операционная система выделяет участок памяти из кучи.

Когда память больше не нужна, программа освобождает её (например, через free).

Куча управляется либо самой программой, либо специальным аллокатором памяти (программным модулем, который следит за тем, какие участки памяти заняты, а какие свободны).

**Как устроена куча?**

Куча — это просто большой непрерывный кусок памяти, который программа получает от операционной системы. Но чтобы управлять этим куском, нужно как-то организовать его. Вот тут и начинается самое интересное.

Как операционная система даёт программе память?

В задаче тебе нужно использовать системный вызов mmap(2), чтобы получить анонимный регион памяти. Давай разберём, что это такое:

*mmap(2)* — это системный вызов в Unix-подобных системах (например, Linux), который позволяет программе запросить у операционной системы кусок памяти.

**Анонимный регион** — это память, которая не привязана к какому-либо файлу на диске. Она просто создаётся "из воздуха" и используется программой.

Когда ты вызываешь mmap, ты говоришь: "Дай мне кусок памяти размером X байт". Операционная система выделяет этот кусок и возвращает указатель (адрес) на его начало.

Пример вызова mmap:

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);addr: обычно NULL (операционная система сама выберет адрес).

length: размер памяти в байтах.

prot: права доступа (например, PROT\_READ | PROT\_WRITE для чтения и записи).

flags: флаги, например MAP\_PRIVATE | MAP\_ANONYMOUS (анонимная память, не связанная с файлом).

fd и offset: для анонимной памяти не нужны, ставим -1 и 0.

После вызова mmap ты получаешь указатель на начало выделенной памяти, и теперь это твоя куча, которой ты будешь управлять.

**Как работает my\_malloc и my\_free?**

**1. my\_malloc(size\_t size)**

Эта функция должна:

- Принять размер памяти в байтах, который нужно выделить.

- Найти в куче свободный участок памяти подходящего размера.

- Пометить этот участок как занятый.

- Вернуть указатель на начало этого участка.

- Если свободного участка нет, вернуть NULL.

Чтобы это сделать, тебе нужно как-то организовать кучу, чтобы знать, какие её части свободны, а какие заняты.

**2. my\_free(void \*ptr)**

Эта функция должна:

- Принять указатель на участок памяти, который был выделен ранее my\_malloc.

- Пометить этот участок как свободный, чтобы его можно было использовать снова.

**Как организовать кучу?**

Чтобы управлять кучей, нужно отслеживать, какие её части заняты, а какие свободны. Для этого обычно используют метаданные — небольшие структуры данных, которые хранят информацию о каждом участке памяти.

Простая структура кучи

Один из способов организовать кучу — разбить её на блоки. Каждый блок состоит из:

Заголовка (header): хранит информацию о блоке (например, его размер и статус — занят или свободен).

Данных: сама память, которую ты выделяешь для программы.

Пример структуры блока:

struct block {

size\_t size; // Размер блока (включая заголовок и данные)

int free; // 1, если блок свободен, 0, если занят

struct block \*next; // Указатель на следующий блок

};Куча будет выглядеть как связный список таких блоков. Когда ты вызываешь my\_malloc, ты ищешь свободный блок подходящего размера. Когда вызываешь my\_free, ты помечаешь блок как свободный.

**Фрагментация памяти**

Теперь поговорим о фрагментации — одной из главных проблем при управлении кучей. Фрагментация возникает, когда память делится на маленькие кусочки, из-за чего сложно найти большой непрерывный участок, даже если в сумме свободной памяти достаточно.

Типы фрагментации

**1. Внешняя фрагментация:**

Это когда свободная память разбита на маленькие кусочки, которые не находятся рядом.

Пример: у тебя есть 100 байт свободной памяти, но она разбита на 10 кусков по 10 байт. Если тебе нужен блок в 50 байт, ты не сможешь его выделить, хотя в сумме памяти хватает.

Проблема внешней фрагментации в том, что память становится "разбросанной".

**2. Внутренняя фрагментация:**

Это когда ты выделяешь больше памяти, чем нужно, из-за ограничений аллокатора.

Пример: твой аллокатор всегда выделяет блоки кратные 16 байтам. Если программе нужно 10 байт, ты всё равно дашь 16, и 6 байт останутся неиспользованными (это и есть внутренняя фрагментация).

**Алгоритмы управления памятью в куче**

Чтобы минимизировать фрагментацию и эффективно управлять кучей, используют разные алгоритмы выделения памяти. Вот самые популярные:

**1. First Fit (Первый подходящий)**

Как работает: Ищешь первый свободный блок, который достаточно велик для запроса.

+ Плюсы:

Прост в реализации.

Быстро работает, так как останавливается на первом подходящем блоке.

- Минусы:

Может приводить к внешней фрагментации, так как большие блоки разбиваются на мелкие.

Часто оставляет маленькие неиспользуемые куски в начале кучи.

**2. Best Fit (Лучший подходящий)**

Как работает: Ищешь свободный блок, который ближе всего по размеру к запросу (минимизируя внутреннюю фрагментацию).

+ Плюсы:

Меньше внутренней фрагментации, так как подбирает блок, максимально близкий к нужному размеру.

- Минусы:

Требует просмотра всех блоков, что медленнее.

Может создавать больше внешней фрагментации, так как оставляет мелкие "остатки".

**3. Worst Fit (Худший подходящий)**

Как работает: Выбираешь самый большой свободный блок, чтобы минимизировать вероятность, что он станет слишком маленьким для будущих запросов.

+ Плюсы:

Может уменьшить внешнюю фрагментацию в некоторых случаях.

- Минусы:

Часто приводит к внутренней фрагментации, так как использует слишком большие блоки.

Медленнее, так как требует поиска самого большого блока.

**4. Next Fit (Следующий подходящий)**

Как работает: Похож на First Fit, но поиск начинается с того места, где закончился последний выделенный блок.

+ Плюсы:

Быстрее, чем Best Fit, так как не просматривает всю кучу.

Может уменьшить фрагментацию в начале кучи.

- Минусы:

Может приводить к неравномерному использованию памяти.

**5. Buddy System (Система парного выделения)**

Как работает: Куча делится на блоки, размер которых — степень двойки (например, 16, 32, 64 байта). Если нужен блок определённого размера, большой блок делится пополам, пока не получится подходящий размер.

+ Плюсы:

Простое объединение блоков (слияние соседних свободных блоков).

Уменьшает внешнюю фрагментацию.

- Минусы:

Высокая внутренняя фрагментация, так как выделяет блоки только фиксированных размеров.

**Современные системы управления памятью**

В современных операционных системах и языках программирования используются более сложные аллокаторы, которые комбинируют разные подходы, чтобы минимизировать фрагментацию и повысить производительность. Вот несколько примеров:

**1. glibc malloc (Linux)**

Используется в стандартной библиотеке C в Linux.

Комбинирует разные алгоритмы:

Для маленьких блоков использует пулы памяти (bins), где хранятся блоки фиксированного размера.

Для больших блоков применяет что-то вроде Best Fit.

Использует mmap для очень больших запросов (например, больше 128 КБ).

Уменьшает фрагментацию за счёт объединения соседних свободных блоков (coalescing).

**2. jemalloc**

Используется в FreeBSD, Firefox, и многих других проектах.

Оптимизирован для многопоточных приложений.

Использует пулы памяти и арены (отдельные кучи для каждого потока), чтобы избежать конкуренции между потоками.

Минимизирует фрагментацию с помощью сложных алгоритмов, таких как Best Fit с учётом выравнивания.

**3. tcmalloc (Google)**

Разработан Google для высокой производительности.

Использует кэширование освобождённой памяти для каждого потока, чтобы ускорить выделение.

Минимизирует фрагментацию за счёт разделения памяти на маленькие и большие блоки.

**4. Slab Allocator**

Используется в ядре Linux для управления памятью.

Делит память на слабы (группы блоков одного размера), чтобы минимизировать фрагментацию.

Подходит для объектов фиксированного размера (например, структур в ядре).

