Лекция 05 — Работа с памятью

”Я опять ничего не понял.”
— П. С. Скаков

Процессы и потоки

Одна из основных задач операционной системы — распределение аппаратных ресурсов (в частности ресурсов процессора и памяти) между исполняемыми программами, т.е. их **аллокация**.
При описании этого процесса традиционно выделяется две абстракции:

> ⓘ **Info**
>
> **Процесс** — сущность ОС, объединяющая в себе все ресурсы, необходимые для выполнения программы.

> ⓘ **Info**
>
> **Поток** — сущность ОС, выполняющая обработку программного кода.
>
> В современной архитектуре поток является основным составным компонентом процесса.

> ⚠ **Warning**
>
> Все потоки внутри одного процесса разделяют его ресурсы.

В момент запуска бинарного исполняемого файла ОС создаёт новый процесс (обычно дочерний процесс оболочки) и главный (он же *первый*, *первичный*) поток в нём, с которого начинается исполнение программы.

Виртуальное адресное пространство

Непосредственное предоставление процессу физических адресов для работы с памятью чревато.
Например, в такой модели затруднительно как-либо разграничивать блоки памяти между собой, а значит процессы могут и будут мешать работе не только друг друга, но и операционной системы.
Для решения этой и ряда других проблем вводится абстракция на основе физической памяти — **адресное пространство**.

> **Info**
>
> **Адресное пространство** — набор адресов, которые могут быть использованы процессом для обращения к памяти.

Более подробно о процессе назначения программе её адресного пространства вы узнаете во втором семестре на курсе архитектуры электронно-вычислительных машин. Сейчас нам важно лишь понимать, что адресные пространства в общем случае не пересекаются, гарантируя сохранность данных и стабильность системы.

Другая проблема, возникающая при аллокации памяти, заключается в значительной ограниченности объёма физических накопителей. Она стоит особенно остро, если мы хотим параллельно запускать несколько ресурсоёмких процессов.

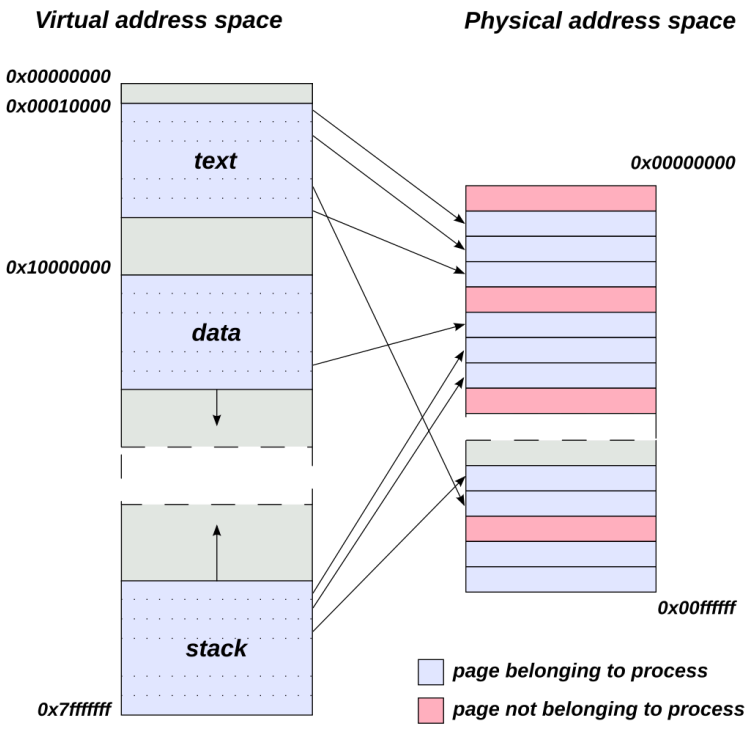
> **Info**
>
> Одним из решений является механизм **виртуальной памяти**, реализующий две основных концепции:
>
> 1. каждому процессу сопоставляется изолированное *виртуальное* адресное пространство, адреса которого специальным образом транслируются в физические адреса, при этом программа получает иллюзию единоличного владения всем доступным объёмом памяти;
> 2. объём ОЗУ прозрачно для программы расширяется путём выгрузки неактивных участков памяти в специальную область на диске (область подкачки).
>
> В контексте данной темы мы будем рассматривать преимущественно первую из этих идей.

> ⚠ **Warning**
>
> Объём виртуальной памяти ограничен размером накопителя, используемого для хранения файла подкачки.

Трансляция адресов

> ⓘ **Info**
>
> Виртуальное адресное пространство состоит из блоков фиксированного размера, называемых **страницами**. Соответствующие блоки в физической памяти называются **страничными блоками**.
>
> Размеры страниц и страничных блоков обычно совпадают и равны 4 КБ.

> ⓘ **Info**
>
> ОС хранит сопоставление между виртуальным и физическим адресом в особой структуре данных, называемой **таблица страниц** (*page table*).



Механизм трансляции. Segmentation fault

> **Info**
>
> Компонент процессора, реализующий перевод виртуального адреса в физический, называется **блоком управления памятью** (*Memory Management Unit*, *MMU*).

MMU хранит *буфер ассоциативной трансляции* (*Translation Lookaside Buffer*, *TLB*) — кэш недавно использованных отображений из таблицы страниц. При каждом новом запросе на перевод виртуальной памяти в физическую в первую очередь просматривается именно TLB. Если нужное соответствие нашлось в кэше, то нужный физический адрес возвращается. В противном случае мы начинаем просмотр таблицы страниц, и как только мы обнаруживаем нужное соответствие, оно заносится в буфер, к которому мы в дальнейшем повторно обращаемся.

Если поиск соответствия завершается неудачей, то MMU возвращает исключение, известное как *page fault*. Это может произойти по двум причинам:

1. Необходимая страница отсутствует в памяти. Это может быть как легитимная ситуация, при которой нужная память выгружена на диск, так и ошибка обращения по невалидному адресу.
2. Страница присутствует в памяти, но мы не обладаем достаточными правами доступа к ней.

Когда ядро ОС сталкивается с этим исключением, оно анализирует его причину.
В случае, если страница хранится в области подкачки, оно выгрузит на диск какое-то количество неиспользуемых на данный момент страниц, вернет запрошенные данные в оперативную память и вернет его пользователю.
Во противном случае оно аварийно завершит проблемный процесс, отправив ему соответствующий сигнал или исключение, который мы, в свою очередь, воспримем как **ошибку сегментации**.

> **Warning**
>
> С практической точки зрения, к ошибке сегментации ведут следующие действия:
>
> - обращение по нулевому или неинициализированному указателю;
> - обращение по указателю на несуществующий адрес;
> - попытка записи в read-only память;
> - переполнение стека или кучи и проч.

Структура виртуального адресного пространства

Всё виртуальное адресное пространство разбито на две области:

1. **Kernel Space** (пространство ядра) — привилегированная область памяти, зарезервированная под системные файлы и защищенная от пользовательских программ.
2. **User Space** (пользовательское пространство) — область памяти, в которой выполняются пользовательские приложения.

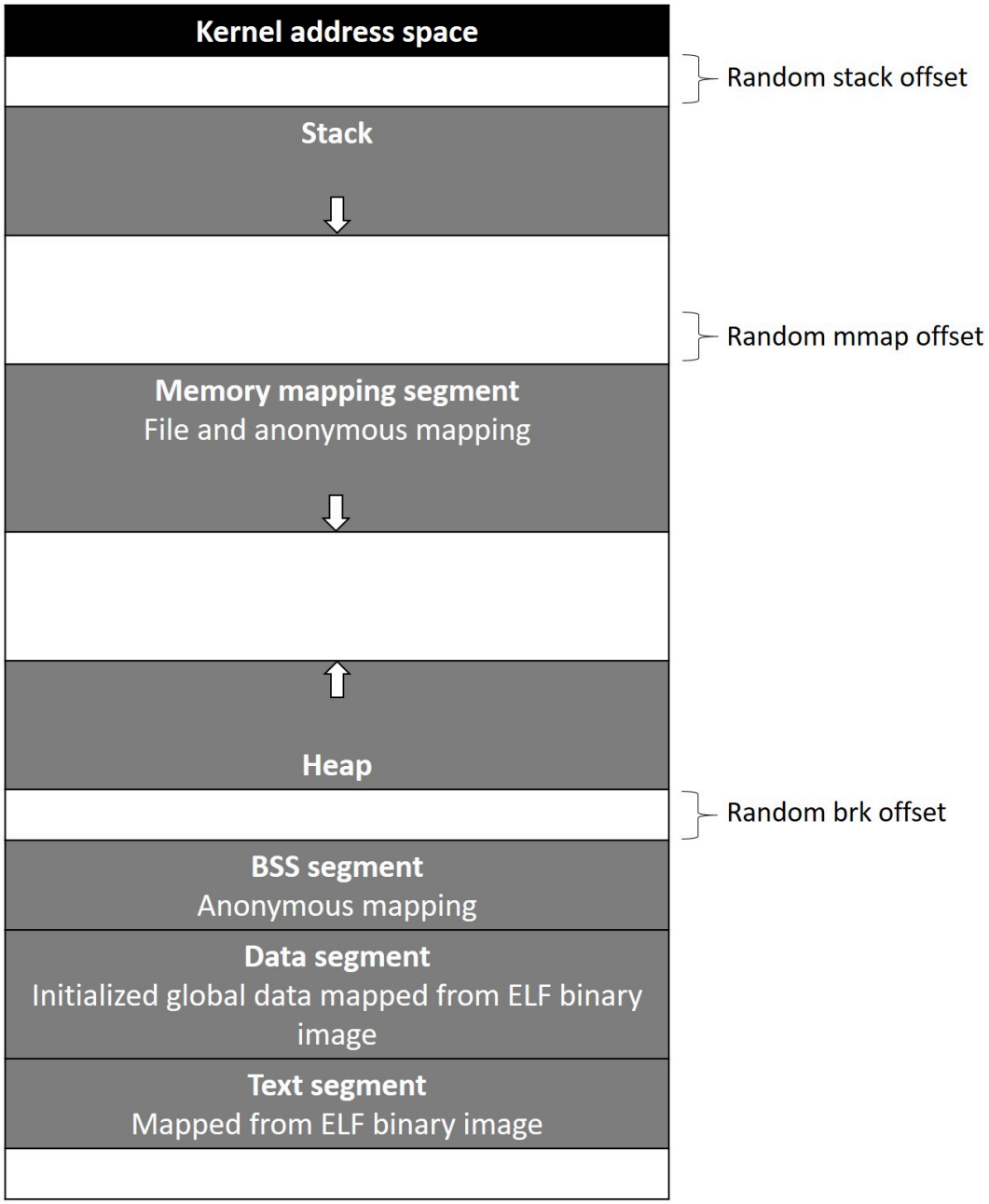
Пользовательское пространство, в свою очередь, разбито на сегменты на основании семантики различных областей памяти:

Сегмент	Хранимые сведения	Права доступа	Особенности
Text	Исполняемый машинный код программы	`r-x`	Read-only память, не изменяется во время выполнения
Data	Инициализированные глобальные и статические переменные	`rw-`	Значения известны при компиляции, хранятся в исполняемом файле
BSS	Неинициализированные глобальные и статические переменные	`rw-`	В исходнике хранится только размер, при загрузке ОС заполняет нулями
Heap	Динамически выделяемая память	`rw-`	Растет в сторону старших адресов, управляется программистом/сборщиком мусора
Stack	Локальные переменные, аргументы функций, контекст вызовов	`rw-`	Растет в сторону младших адресов, управляется автоматически, ограниченный размер
Memory Mapping	Файлы, библиотеки, большие блоки памяти	`rw-` / `r-x`	Универсальная область, может расти в обе стороны, поддерживает разделение между процессами

Все сегменты пользовательского пространства можно разбить на две группы:

1. *статические* (`Text`, `Data`, `BSS`), размер и положение которых не изменяется в ходе выполнения программы;
2. *динамические* (`Stack`, `Heap`, `Memory Mapping`), размер и положение которых может изменяться в ходе выполнения программы.

Так, всё виртуальное адресное пространство целиком имеет приблизительно следующую структуру[1]:



Практическое исследование

При помощи средств языка C++ и нашей ОС мы можем непосредственно посмотреть на то, как организованы сегменты памяти внутри виртуального адресного пространства. Рассмотрим следующий код:

```
#include <iostream>
#include <unistd.h>

static const float global_const = 3.14f;
static int uninitialized_var;
int global_var = 238;

void func() {
    return;
};

int main() {
    int local_var = 0;
    char* str = "Hello, World!\n";

    std::cout << "Process ID: " << getpid() << '\n';

    std::printf("Address of global_const: %p\n", &global_const);
    std::printf("Address of global_var: %p\n", &global_var);
    std::printf("Address of str: %p\n", &str);
    std::printf("Address of func: %p\n", &func);
    std::printf("Address of local_var: %p\n", &local_var);
    std::printf("Address of uninitialized_var: %p\n", &uninitialized_var);

    getchar(); // необходимо, чтобы процесс не завершился
    return 0;
}
```

После компиляции в стандартный поток выведется идентификатор процесса и адреса́ соответствующих объектов в виртуальном адресном пространстве.
Поскольку процесс после этого останется незавершенным, мы можем обратиться к нему и, посмотрев на маппинги, проанализировать расположение как объектов относительно ВАП, так и диапазонов ВАП относительно физической памяти. В Linux это можно сделать при помощи консольной команды `cat /proc/<process_id>/maps`.

Крайне рекомендуется проделать эти манипуляции самостоятельно, в процессе экспериментируя с типами переменных и отслеживая их расположение в памяти.

Стек вызовов

Info

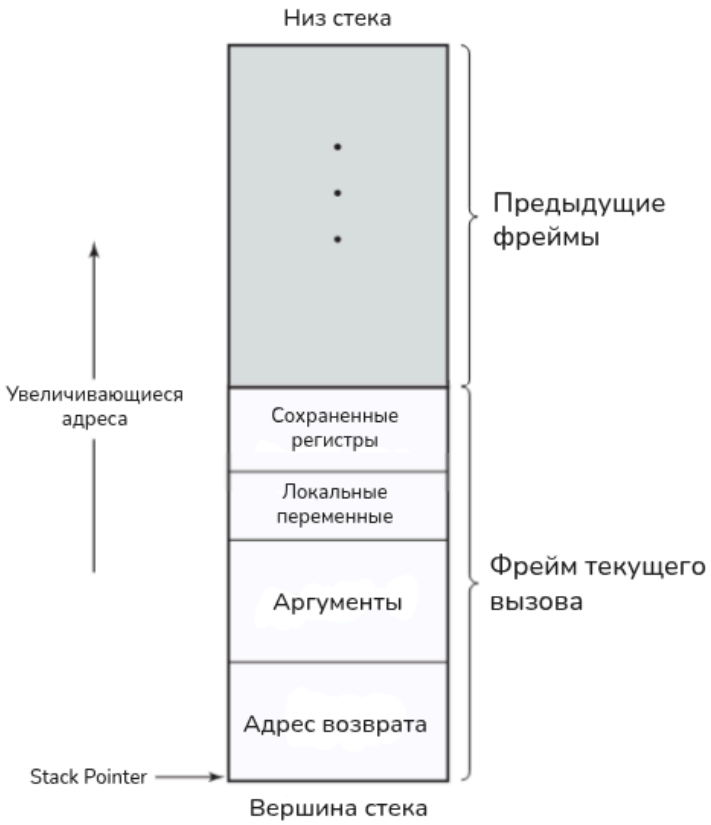
Стек вызовов — динамический сегмент пользовательского пространства виртуальной памяти, реализующий механизм вызова функций.

Info

Основным элементом стека является **фрейм вызова**, хранящий контекст выполнения конкретной функции в соответствующей цепочке. В этот контекст входят:

- аргументы функции
- локальные переменные
- сохраненные регистры процессора
- адрес возврата

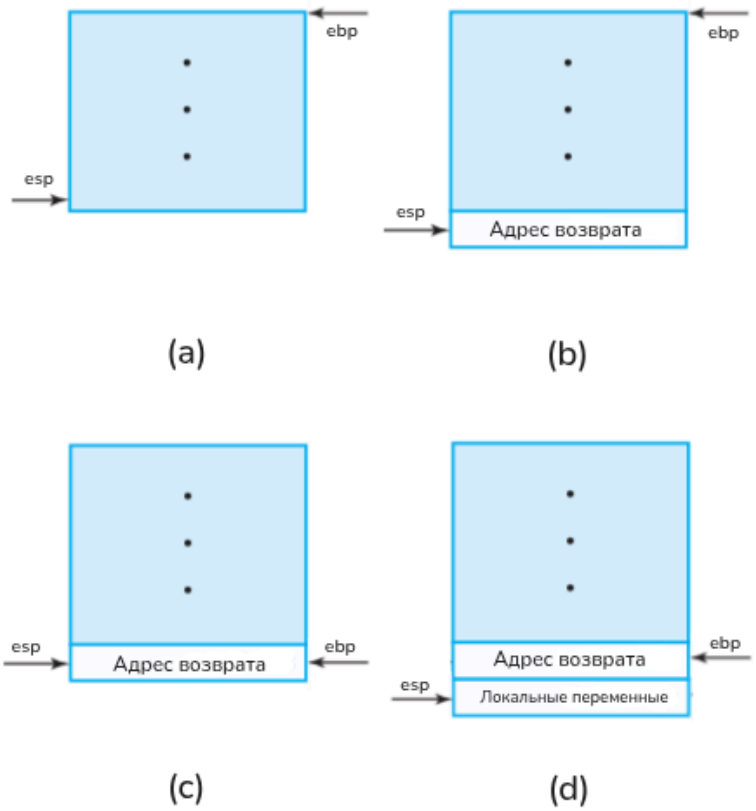
Перед первым фреймом в стеке записана системная информация об аргументах запуска процесса, переменных среды и проч.



Механизм работы (на примере архитектуры x86-64)

Когда информация о новом вызове добавляется в стек, происходит следующее:

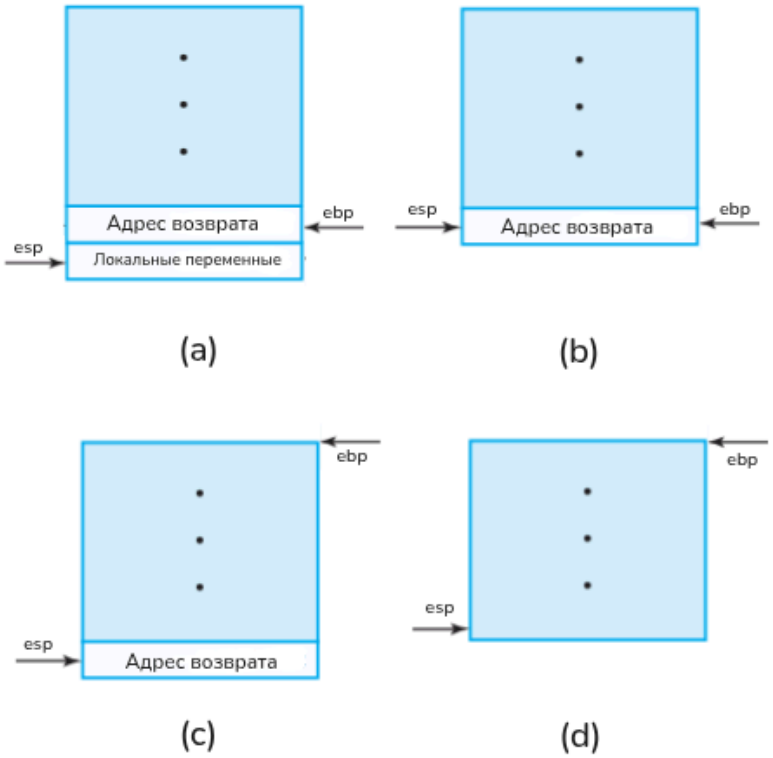
1. сохраняется адрес возврата;
2. устанавливается указатель на начало нового фрейма;
3. указатель на вершину стека сдвигается, выделяя память под локальные переменные.



Процесс добавления нового стекфрейма:
(a) стек до вызова новой функции
(b) сохранение адреса возврата
(c) перемещение указателя на текущий фрейм
(d) выделение памяти под локальные переменные

Когда соответствующий вызов оказывается обработан, происходит следующее:

1. восстанавливается указатель на вершину стека;
2. восстанавливается указатель на начало предыдущего фрейма;
3. вычисленное значение передаётся по адресу возврата[2].



Процесс удаления стекфрейма:
(a) стек до выхода из функции
(b) восстановление указателя на вершину стека
(c) восстановление указателя на начало предыдущего фрейма
(d) стек после возврата значения

Этими процессами управляют два регистра процессора:

- `esp` / `rsp` (Stack Pointer) — указатель на вершину стека
- `ebp` / `rbp` (Frame Pointer) — указатель на начало последнего фрейма

Так, в терминах машинного кода взаимодействие со стеком сводится к следующему набору инструкций:

```
; Вход в функцию
push ebp            ; Сохраняем предыдущий ebp
mov ebp, esp        ; Устанавливаем новый ebp
sub esp, N          ; Выделяем место для локальных переменных путем сдвига
esp

; Выход из функции
mov esp, ebp        ; Восстанавливаем esp
pop ebp             ; Восстанавливаем предыдущий ebp
ret                 ; Возвращаем вычисленное значение
```

Как видно из этого алгоритма, стек расширяется "вниз", то есть чем позже выделена память, тем меньший адрес она имеет.

> ⚠ **Warning**
>
> Все из этих команд генерируются компилятором — это означает, что управление памятью в стеке *автоматическое*, то есть в процессе написания программы мы полностью свободны от необходимости вручную контролировать процессы выделения и освобождения памяти.

Метрики и переполнение стека

Доступ к стеку — быстрая операция. Это достигается рядом факторов, в числе которых, например, простота самого механизма или частое переиспользование одних и тех же байтов, позволяющее предположить, что в реальности они почти всегда хранятся в кэше процессора.

К сожалению, это быстродействие во многом завязано на локализации данных и, соответственно, небольшом размере самого стека — чаще всего он не превышает 1 МБ. Главная проблема, которая возникает из-за этого ограничения — это переполнение стека, которое происходит при чрезмерном объёме данных или слишком глубокой рекурсии. Читателю предлагается самостоятельно понять, почему этого лучше не допускать.

Динамические аллокации. Куча

Иногда нам приходится сталкиваться с ситуациями, когда размер определенной структуры данных неизвестен программе вплоть до момента её запуска. В таком случае можно обойтись достаточно большой константой, определяющей максимальный возможный размер, однако у этого подхода есть серьёзная проблема: заданное нами значение совершенно никак не соотносится с реальным объёмом памяти на машине, которая будет исполнять код. Если мы, например, вдруг захотим работать с данными большего размера, то единственным решением будет полностью перекомпилировать программу с другим значением константы, что порой может вызывать серьезные затруднения.

Именно здесь возникает концепция *динамических аллокаций*. Задача проста — каким-то образом научиться выделять память, отказавшись от hard-coded констант и основываясь только на данных, которые программа получает в процессе своей работы. В определённом смысле её успешно решает стек, однако выше мы уже обозначили его

главную проблему — крайне ограниченный объём реальной памяти, который не позволяет нам использовать его для хранения структур большого размера. По этой причине мы выделяем под решение этой задачи отдельный сегмент, называемый **кучей**.

> **Info**
>
> **Куча** — динамический сегмент пользовательского пространства виртуальной памяти, реализующий механизм динамической аллокации.

Аллокаторы

Хотя аллоцировать память можно вручную при помощи низкоуровневых средств, чаще всего эту работу делает специальный интерфейс — *аллокатор*.
Он воспринимает кучу как некоторое количество подряд идущих блоков памяти произвольного размера, каждый из которых либо аллоцирован, либо свободен, и реализует алгоритмы заполнения и освобождения этих блоков.

> **Warning**
>
> Любой аллокатор работает в рамках следующих ограничений:
>
> 1. последовательности запросов могут быть произвольными;
> 2. запросы должны обрабатываться незамедлительно;
> 3. блоки памяти должны быть выровнены;
> 4. модифицировать аллоцированные блоки памяти запрещено.

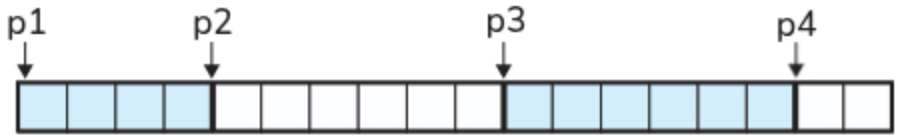
Одна из главных задач аллокатора — утилизировать память настолько оптимально, насколько это возможно. В связи с этим совершенно естественным кажется желание уметь не только аллоцировать память, но и *деаллоцировать*, т.е. освобождать ненужные блоки для дальнейшей перезаписи. Как только мы забываем про деаллокацию, происходит то, что называется **утечкой памяти** — если каждый раз выделять новую память, не переиспользуя старую, то в конечном итоге её количество будет неизбежно уменьшаться, что очень часто может быть критичным.

> **Warning**
>
> В отличие от стека, управление памятью в куче *ручное*. Мы должны самостоятельно следить за тем, чтобы не допускать утечек памяти.

Фрагментация [3]

Риск утечки памяти является не единственной и не главной проблемой динамических аллокаций.

Рассмотрим абстрактную кучу, которая в результате выполнения какой-то последовательности запросов приняла следующий вид:



Здесь p1, p2, p3 и p4 — указатели на соответствующие блоки памяти.
Аллоцированные блоки выделены голубым цветом, размер ячейки составляет 1 байт.

Нетрудно заметить, что количество свободной памяти в куче на данный момент составляет 8 байт. Точно так же нетрудно заметить, что как бы мы ни старались, выделить эти 8 байт без расширения самой кучи не представляется возможным: ни один из свободных блоков не имеет достаточного размера, при этом увеличить его, оставаясь в рамках текущей памяти, мы не в состоянии, поскольку для этого необходимо было бы воздействовать на занятые блоки, на что аллокатор не имеет права.

> **Info**
>
> Если аллоцированные блоки расположены непоследовательно, то они разбивают всё свободное пространство на разрозненные участки, размер каждого из которых может оказаться меньше требуемого для новой аллокации, даже если суммарно свободной памяти достаточно. Эта проблема называется проблемой **внешней фрагментации** памяти.

> **Info**
>
> Помимо того, что участки свободной памяти могут располагаться непоследовательно, какая-то её часть также может находиться непосредственно внутри частично заполненных блоков и, соответственно, не быть доступной для использования. Такое поведение зависит от реализации конкретного аллокатора и происходит, например, в результате выравнивания или хранения метаданных. Эта проблема называется проблемой **внутренней фрагментации** памяти.

В то время как внутренняя фрагментация является детерминированным процессом, зависящим только от реализации аллокатора и предыдущих запросов к нему, объём внешней фрагментации гораздо сложнее оценить и невозможно предсказать, поскольку

на неё оказывают влияние не только предыдущие запросы, но и будущие. По этой причине большинство аллокаторов стремятся применять различные эвристические оптимизации, чтобы свести ущерб от фрагментации к минимуму.

Механизм аллокации

Рассмотрим главные этапы аллокации и основные практические подходы к ним.

> **1. Выравнивание**
>
> Многие процессоры и инструкции требуют выравнивания для безопасной и/или эффективной работы. Поскольку оно является одной из причин внутренней фрагментации, то в определенном смысле мы жертвуем оптимальным распределением памяти взамен на высокую производительность.

> **2. Поиск подходящего блока**
>
> На концептуальном уровне есть несколько стратегий, решающих эту задачу:
>
> - **First Fit** — самый примитивный подход, подрезумевающий, что мы всегда выбираем первый подходящий по размеру блок. Он достаточно быстрый, однако в общем случае плохо справляется с фрагментацией, и поэтому используется только в самых простых аллокаторах или как часть гибридных подходов.
> - **Next Fit** — подход, подразумевающий циклический поиск нужного блока, начинающийся с места последнего успешного выделения. Он может работать значительно быстрее First Fit, но несколько хуже утилизировать память.
> - **Best Fit** — подход, подразумевающий просмотр всей кучи и поиск блока наименьшего подходящего размера. В стандартной реализации он медленен, однако всегда гарантирует наименьшую внешнюю фрагментацию.
> - **Worst Fit** — подход, обратный к Best Fit и подразумевающий поиск наибольшего подходящего блока. Это специфичный алгоритм, который используется только в особых случаях.

> **3. Выделение блока**
>
> Если аллокатор не находит подходящий блок, то он направляет ОС запрос на выделение дополнительной памяти, и, если этот запрос был удовлетворён, работает уже с ней. Здесь также следует заметить, что куча расширяется "вверх", т.е. чем позже выделена память, тем больший адрес она имеет.
>
> Как только подходящий блок оказывается найден, аллокатору необходимо принять решение о том, какую его часть выделить под запрос. Выделение всего блока рано или поздно приведет ко внутренней фрагментации, так как его размер может быть

> значительно больше реального необходимого объёма, и хотя это может быть приемлемо при определенных обстоятельствах, чаще всего нам хочется исключить такое поведение. По этой причине имеет смысл *разделить* блок на две части: аллоцируемую и незанятую, которая станет новым свободным блоком, и только после этого выделить запрошенную память.

Деаллокация и coalescing

Идейно процесс деаллокации крайне примитивен: мы уже знаем адрес блока и всю метаинформацию о нем, и наша задача — просто освободить его.

Тем не менее, при частых деаллокациях возникает следующая проблема: когда мы освобождаем подряд идущие блоки, образуется большой свободный пласт памяти, который воспринимается аллокатором как разные фрагменты и не может быть целиком выделен под нужды процесса — это также называется *ложной фрагментацией*.

Решение у данной проблемы также весьма прозаичное: мы просто должны объединять такие блоки в один. Процесс этого объединения называется **coalescing**.
Мы можем выполнять его как сразу при освобождении каждого блока (*immediate coalescing*), либо в какой-то произвольный момент после этого (*deferred coalescing*).

Структура блоков

Как видно из описанного выше, бóльшая часть действий аллокатора сводится к итерации по блокам и произведению манипуляций с ними. По этой причине нам критически важно, чтобы эти процессы были организованы достаточно эффективно.

Info

Реализация отдельного блока как структуры включает в себя:

- заголовок с метаданными (*Header*)
- аллоцированные данные
- паддинг (если необходимо)
- копию заголовка (*Footer*)

Общая структура блока
(с использованием boundary tags)

Заголовок представляет собой одно число, в котором мы хотим хранить размер блока (в байтах) и дополнительные сведения. Благодаря выравниванию размер блока всегда будет кратен 8 байтам, то есть первые 3 бита размера всегда будут равны 0. Поскольку они не несут в себе никакой информации, мы можем записать в них любые необходимые нам метаданные.

Когда мы производим coalescing, нам может понадобиться объединить текущий блок как со следующим, так и с предыдущим. Тогда как при любом методе организации блоков у нас будет возможность непосредственно сослаться на следующий из них, поиск предыдущего может занимать время, линейное относительно размера кучи. Процесс можно оптимизировать, если дублировать заголовок в конец блока — так мы одним смещением указателя сможем получить всю информацию о предыдущем фрагменте памяти. Эта техника называется *boundary tags*[4].

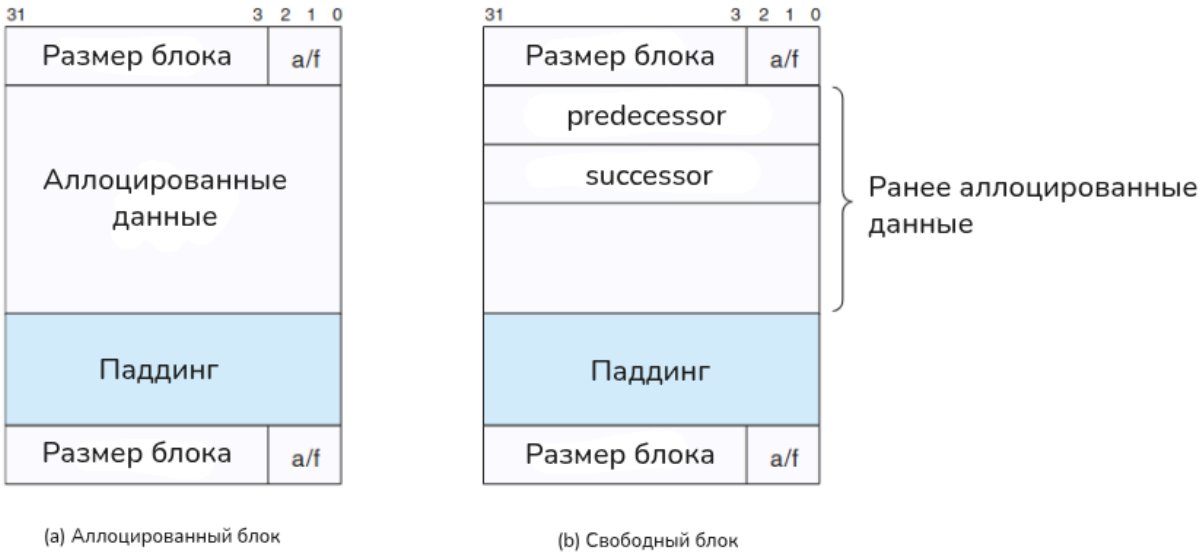
Рассмотрим три способа организовать взаимодействие между блоками:

> **1. Неявный связный список** (Implicit Free List)
>
> Поскольку блоки расположены в памяти последовательно, то, зная адрес и размер текущего блока, мы всегда можем получить адрес следующего. Таким образом, множество блоков с метаданными само по себе является связным списком, причем ссылка на следующий элемент хранится неявно.
> Главная проблема этого подхода в том, что хотя его название и содержит в себе слово *free*, на самом деле мы не умеем рассматривать только свободные блоки, ибо не можем изменять значение неявной ссылки — она всегда указывает на следующий

блок вне зависимости от того, аллоцирован ли он.
Таким образом, аллокация занимает время, линейное относительно общего количества блоков.

2. Явный связный список (Explicit Free List)

Поскольку мы хотим улучшить время аллокации до линейного относительно количества *свободных* блоков, то логичным решением будет попытаться организовать все такие блоки в какую-либо явную структуру данных.
Поскольку программа по определению не запрашивает данные внутри пустого блока, мы можем использовать эту память для того, чтобы хранить указатели на предыдущий (*pred*, *predecessor*) и следующий (*succ*, *successor*) пустые блоки. Формат хранения аллоцированных блоков при этом остается неизменным:



Если мы имеем такую структуру, время аллокации становится линейным относительно количества свободных блоков, поскольку теперь мы можем итерироваться только по ним.

Порядок свободных блоков в связном списке *не обязан* соответствовать их порядку в куче. Так, его можно задать следующими способами:

1. Принцип **LIFO** — каждый освобожденный блок добавляется в начало списка.
2. Принцип **FIFO** — каждый освобожденный блок добавляется в конец списка.
3. Порядок по **адресу**: адрес каждого блока меньше, чем адрес следующего.
4. Порядок по **размеру**: размер каждого блока не меньше (не больше), чем размер следующего.

Каждый из этих порядков будет по-разному сказываться на времени работы и на утилизации памяти.

Главный недостаток такого подхода — внушительный минимальный размер пустого блока. Помимо этого, хоть мы и добились значительного ускорения сравнительно неявного связного списка, такое представление блоков всё ещё не является самым эффективным.

3. Сегрегированный связный список (Segregated Free List)

Произвольно разобъём множество всех возможных размеров блоков на классы эквивалентности (они же *bins*) — например, на диапазоны по степеням двойки:

$$\{1\}, \{2\}, \{3, 4\}, \{5 - 8\}, \ldots, \{1025 - 2048\}, \{2049 - 4096\}, \{4096 - \infty\}$$

Основная идея в том, чтобы завести для каждого класса собственный связный список. Если мы сделаем это, то получим возможность крайне быстро аллоцировать, поскольку изначально сможем рассматривать свободные блоки только подходящего размера. Это также позволяет нам реализовывать гибридные подходы, по-разному работающие с блоками различных размеров. При всём этом мы практически не теряем в производительности остальных операций.

На данном этапе (а на самом деле уже на этапе разбиения на классы) существует масса способов по-разному реализовать сегрегированный связный список. В качестве примера рассмотрим один из наиболее широко используемых.

Segregated Fits

Будем поддерживать массив связных списков (явных или неявных), соответствующих классам размерности.
Чтобы аллоцировать блок памяти какой-то длины, мы определяем его класс и выполняем поиск подходящего фрагмента в соответствующем массиве. Если поиск безуспешен, то мы переходим к следующему по возрастанию размерности классу. Если в итоге не нашлось ни одного подходящего блока, то мы запрашиваем память у ОС.
Как только нужный блок был найден, мы разделяем его и добавляем пустой остаток к соответствующему его размерности классу.
При удалении блока мы выполняем coalescing и также добавляем результат к соответствующему классу.

Этот подход эффективен как с точки зрения времени работы, поскольку мы пропускаем множество лишних блоков, так и с точки зрения утилизации памяти, поскольку практически мы реализуем идею best-fit поиска.

Аллокаторы стандартной библиотеки

Стандартная библиотека языка C предоставляет собственную реализацию интерфейса аллокатора — функцию `malloc()`.
Она принимает на вход единственное значение `N` — объём аллоцируемой памяти в байтах. Если аллокация успешна, то `malloc()` вернёт указатель на первую свободную ячейку выделенного блока памяти; иначе вернётся `NULL`.

Взаимодействие `malloc` с ОС реализовано при помощи двух методов: `sbrk()` — расширение сегмента кучи — и `mmap()` — выделение страницы памяти в сегменте Memory Mapping.

Категории классов

Инженерия блоков в `malloc` расширяет концепцию Segregated Fits. Все пустые блоки делятся на несколько категорий классов, для каждой из которых аллокатор реализует уникальную логику:

> **Fast Bins** (7/10) — 16, 24, 32, 40, 48, 56 и 64[5] байт соответственно.
>
> Это классы блоков фиксированного маленького размера, доступ к которым должен быть максимально быстрым. Из этих соображений мы откладываем их объединение, выполняя его на других стадиях аллокации.
> Блоки организованы по принципу *LIFO*. Наряду с отсутствием мгновенного coalescing, это также освобождает нас от необходимости уметь итерироваться по блокам в обе стороны, поэтому fast bins реализуют *односвязные* списки.

> **Small Bins** (62) — до 512 байт.
>
> Это классы блоков фиксированного маленького размера, для которых мы уже не делаем тех оптимизаций, что делали для fast bins. В частности, это означает, что мы всегда сразу выполняем coalescing при освобождении блока, то есть мы также должны уметь удалять произвольный блок, для чего необходимо использовать *двусвязные* списки. Блоки размещаются по принципу *FIFO*, в следствие чего мы минимально их переиспользуем.

> **Large Bins** (62) — более 512 байт.
>
> Это классы больших блоков нефиксированного размера. Так, размер блоков в первых 32 классах совпадает с точностью до 64 байт, в следующих 16 — с точностью до 512 байт, и так далее. Поскольку размер блоков внутри класса варьируется, их необходимо поддерживать в отсортированном *по невозрастанию размера* состоянии,

> что сразу приводит к необходимости использования *двусвязного* списка. Для этих блоков мы также сразу проводим coalescing.

> **Unsorted Bin** (1)
>
> Это единый буфер всех недавно освобожденных блоков. Прежде чем блок попадет в соответствующий ему класс, мы размещаем его здесь. Это позволит оперативно переиспользовать его при следующем вызове `malloc()`, если он удовлетворяет соответствующему запросу.
> Другой неочевидный плюс в том, что вся логика сортировки по small и large классам выносится в часть алгоритма, реализующую обработку `unsorted bin`.
> Поскольку мы должны уметь удалять произвольный блок, когда вносим его в нужный класс, то нам необходимо использовать *двусвязный* список.

Многопоточность и арены

До сих пор мы рассматривали системы, которые обслуживали только один поток.
Если потоков станет несколько, то они начнут конкурировать за память на куче, что может привести к фатальным ошибкам. Во избежание этого `malloc` реализует концепцию арен — непересекающихся областей памяти, хранящих полный контекст управления кучей.

За каждым потоком закрепляется арена, в рамках которой он может запрашивать память. Число арен при этом ограничено — это значит, что несколько потоков всё ещё могут в какой-то момент попытаться обратиться к одному участку памяти. Чтобы избежать конфликтов, при входе в арену соответствующий поток блокирует её от остальных. Если же арена оказывается заблокирована на момент обращения, поток оказывается вынужден ожидать, пока она освободится.

Чтобы минимизировать количество обращений к арене и, как следствие, время ожидания, каждому потоку назначается собственный кэш `tcache`, схожий по структуре с `fast bin`.
Прежде, чем обратиться к арене и заблокировать её, поток сначала пытается выделить память из собственного кэша, и только при неудаче начинаются привычные процедуры аллокации.

По ряду причин подробности реализации системы арен намеренно опускаются. Особо любопытные читатели могут найти материалы по этой теме в списке источников.

Механизм работы `malloc()`

Здесь стоит сделать крайне важную ремарку о том, что, на самом деле, найти достоверную информацию об алгоритме работы `glibc malloc` где-либо, кроме исходного кода — задача не из лёгких. Конечно, ряд источников предоставляет её, однако многие детали между ними разнятся. В связи с этим следует полагать, что приведенный

ниже порядок действий имеет скорее ориентировочный характер, нежели строгий. Впрочем, он всё ещё достаточно близок к истине.

Important

Общий алгоритм действий `malloc()` следующий:

1. Выравниваем размер запроса.
2. Если размер запроса превышает значение `M_MMAP_THRESHOLD` (по умолчанию 128 килобайт), то мы обращаемся к системным средствам маппинга памяти при помощи `mmap()`.
3. Если в `tcache` есть блок нужного размера, выделяем его.
4. Если размер запроса не превышает 64 байта, мы ищем подходящий блок строго в соответствующем `fast bin`. Если поиск успешен, выделяем эту память. Если поиск безуспешен, то поиск в бóльших `fast bins` не выполняется.
5. Если размер запроса не превышает 512 байт, мы ищем подходящий блок во всех `small bins`, в которых он потенциально может лежать. Если поиск успешен, разделяем блок и выделяем память.
6. Консолидируем все `fast bins`, т.е. переносим все блоки оттуда в `unsorted bin`.
7. Сортируем `unsorted bin`, выполняя при этом coalescing. Если в процессе был найден блок подходящего размера, разделяем его и выделяем память.
8. Ищем подходящий блок во всех `large bins`, в которых он потенциально может лежать. Если поиск успешен, разделяем блок и выделяем память.
9. Если на вершине кучи осталось недостаточно памяти, направим системе запрос на расширение. Если теперь памяти достаточно, то разделим соответствующий блок и выделим нужный отрезок памяти.

Механизм работы `free()`

Стандартная библиотека также предоставляет нам собственный инструмент для деаллокаций — функцию `free()`.
`free()` принимает единственное значение — указатель на блок памяти, который необходимо освободить, — и ничего не возвращает.

> ⚠ **Warning**
>
> Указатель, передаваемый в `free()`, **обязан** быть получен в результате вызова `malloc()`, `calloc()` или `realloc()`, при этом эта область памяти **не может** быть ранее деаллоцирована. Нарушение любого из этих условий необратимо приводит к UB.

В определенной степени здесь имеет место аналогичная ремарка, однако стоит отметить, что концептуально вызов `free()` значительно менее сложная процедура, чем вызов `malloc()`. Совершенно не исключено, что приведенный ниже алгоритм может оказаться корректным.

> 🔥 **Important**
>
> Общий алгоритм действий `free()` следующий:
>
> 1. Если блок был выделен через `mmap()`, напрямую вернём эту память системе через `munmap()`.
> 2. Если мы можем сохранить освобожденный блок в кэш потока, сохраним.
> 3. Если мы можем сохранить освобожденный блок в `fast bins`, сохраним.
> 4. Выполним coalescing.
> 5. Если блок не стал вершиной кучи, сохраним его в `unsorted bin`.
> 6. Если на вершине кучи образовался достаточно большой непрерывный блок памяти, то возвратим его системе, физически уменьшив кучу через `brk()`. Этот процесс называется *trimming*.

> ⚠ **Warning**
>
> Из этого алгоритма ясно, что `free()` возвращает память ОС только в двух случаях, причём достаточно редких. Во всех остальных ситуациях мы сохраним блоки памяти в локальные кэши для дальнейшего переиспользования.

Высокоуровневые приложения

Помимо `malloc()` в C/C++ есть ряд других функций, позволяющих аллоцировать память тем же способом, но обладающих несколько отличной семантикой. Так, среди них:

> **calloc()**
>
> Функция `calloc()` принимает на вход два значения: количество `N` и размер `size` объектов, под которые мы хотим выделить память. Если аллокация успешна, то `calloc()` вернёт указатель на первую свободную ячейку выделенного блока памяти; иначе вернётся `nullptr`.
>
> Ключевое семантическое отличие `calloc()` и `malloc()`, помимо сигнатуры, заключается в том, что `calloc()` дополнительно инициализирует выделенную память нулями.

> **realloc()**
>
> Функция `realloc()` реаллоцирует переданную область памяти. Она принимает на вход два значения: указатель `ptr` на начало области реаллокации и размер `new_size`, к которому мы хотим привести эту область. Если реаллокация успешна, то `realloc()` вернёт указатель на первую свободную ячейку выделенного блока памяти; иначе вернётся `nullptr`.
>
> Если `ptr == NULL`, то поведение будет аналогично `malloc(new_size)`. Если `new_size == NULL`, то, начиная с C++23, это необратимо приводит к UB.
>
> Если это возможно, то `realloc()` расширяет либо сужает область памяти, не передвигая её; в противном случае он аллоцирует новый блок памяти размера `new_size`, после чего копирует содержимое исходной области памяти в новую.

> **new/delete**
>
> `new` и `delete` — операторы, добавленные в языке C++ и представляющие собой обёртки аллокаторов из `glibc`. Их сигнатуры семантически совпадают с сигнатурами соответственно `malloc()` и `free()`. Ключевое преимущество `new` и `delete` — поддержка таких концептов, как исключения и перегрузка.
>
> Важно заметить, что в коде `new` и `delete` являются *выражениями* (expression), а не операторами. Когда компилятор встречает одно из них, он сначала вызывает соответствующий оператор, а после — конструктор (деструктор) класса, который мы аллоцировали.

Источники:

1. Э. Таненбаум — Современные операционные системы
2. Randal E. Bryant, David R. O'Hallaron — Computer Systems A Programmer's Perspective
3. Qianying Zhou — Understanding arenas and heaps in malloc() (https://qycode.me/blog/2022/04/Malloc-notes/)
4. sploitfun — Understanding glibc malloc (https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/)
5. wangshuo — Glibc Malloc Principle (https://www.openeuler.org/en/blog/wangshuo/Glibc%20Malloc%20Principle/Glibc_Malloc_Principle)
6. Техническая документация `glibc malloc` (https://sourceware.org/glibc/wiki/MallocInternals)
7. Исходный код `glibc malloc` (https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=994a23248e258501979138f3b07785045a60e69f;hb=HEAD)

---

1. Важно понимать, что точная структура ВАП зависит от системы. Это не столько стандарт, сколько набор архитектурных решений. ↩
2. Инструкция `ret` гарантирует, что в результате её выполнения указатель `esp` будет увеличен на размер адреса возврата, поэтому нам необходимо выполнять никаких дополнительных действий с ним. ↩
3. Фрагментация памяти часто наблюдается у жертв посттравматического стрессового расстройства. ↩
4. Boundary tags, как и, на самом деле, ряд других затрагиваемых здесь концепций, были придуманы Дональдом Кнутом — человеком, который может быть известен читателю как создатель TeX или как одна из фамилий в названии алгоритма Кнута-Морриса-Пратта. ↩
5. В некоторых источниках максимальный размер класса в `fastbin` указан как 80 байт. Читателю предлагается самостоятельно выбрать, во что ему верить. ↩