# Аллокаторы

Николай Пономарев

14 февраля 2026 г.

Санкт-Петербургский
государственный университет

# А теперь системное программирование

- Вы уже имели дело с аллокаторами памяти
- Они нужны, чтобы работать со структурами данных динамического размера
- Аллокатор работает с кучей (heap)
- Для аллокатора куча — множество блоков разного размера
- Блок — непрерывный кусок виртуальной памяти, который может быть либо выделенным (allocated), либо свободным (free)
- НУО предполагаем, что куча растет по увеличению количества адресов

# Какие бывают аллокаторы

## Явные (Explicit)

Программа явно освобождает выделенный блок
`malloc` и `free` в C, `new` и `delete` в C++ ...

## Неявные (Implicit)

Аллокатор сам берет на себя обнаружение и освобождение неиспользованных блоков
Сборщики мусора в Java, .NET, Lisp ...

Мы будем говорить о явных

# Напомним...

`void* malloc(size_t size)`

- выделяет блок памяти размера не менее size
- Return: указатель на выделенный блок или NULL в случае ошибки
- Блок памяти обычно выравнивается (в Unix системах — по 8 байт)
- Не занимается инициализацией! Для этого есть calloc
- В рамках лекции считаем размер слова равным 4 байта

`void free(*ptr)`

- освобождает блок памяти, на который указывает ptr
- ptr должен указывать на начало выделенного блока — иначе UB (если NULL, то ничего не произойдет)
- free не сигнализирует об ошибке — с ним надо быть аккуратнее

# Как аллокатор может выделять память?

- `mmap` и `munmap`
  - Отображение объекта в физической памяти в адресное пространство процесса
  - `/proc/[pid]/maps` — показать отображенные участки памяти процесса
- Управление размером кучи: `void* sbrk(intptr_t incr)`
  - `brk` — указатель на конец кучи
  - `sbrk` просто прибавляет `incr` к этому указателю
  - Надо, чтобы запросить у ОС больше памяти в куче

# Требования к аллокатору

- Последовательность запросов `malloc` и `free` — произвольная
  - Нельзя полагаться на порядок запросов
  - Но мы предполагаем, что `free` вызывается на участке, который был выделен
- Немедленный ответ на запрос
  - Нельзя буферизировать запросы или переупорядочивать
- Используется только куча
  - Все нескалярные структуры данных, которыми пользуется аллокатор, должны лежать в куче
- Выравнивание блоков
  - Нужно, чтобы в блоке могли размещаться данные любого типа
  - В большинстве систем выравнивается по 8 байт
- Нельзя модифицировать выделенные блоки
  - Можно манипулировать только свободными блоками

# Цели аллокатора

## Максимизация пропускной способности (throughput)

Количество запросов, выполняющихся в единицу времени
Нужно уменьшать среднее время на запрос к аллокатору

## Максимальная утилизация памяти (memory utilization)

Полезная нагрузка (payload) — сколько памяти действительно было запрошено
Нам нужно максимизировать суммированную полезную нагрузку для всех запросов относительно размера кучи

Эти цели противоречат друг другу. Нужно искать баланс.

# Фрагментация

Главная причина плохой утилизации кучи
Неиспользованная память не соответствует требованиям запросов аллокатора

## Внутренняя фрагментация

- Выделили больше, чем было запрошено (больше, чем payload)
- Минимальный размер блока
- Выравнивание

## Внешняя фрагментация

- В куче есть место, чтобы выделить память, но нет доступных свободных блоков
- Зависит в том числе от будущих запросов

# Нюансы реализации

Мы могли бы сделать простой аллокатор:

- Куча — массив с указателем p на начало
- `malloc(size)`: увеличить указатель p на size, вернуть новый указатель
- `free(ptr)`: просто return, ничего не делать

Что мы получили:

- Хорошая пропускная способность, все запросы за константу
- Отвратительная утилизация памяти — не переиспользуем свободные блоки

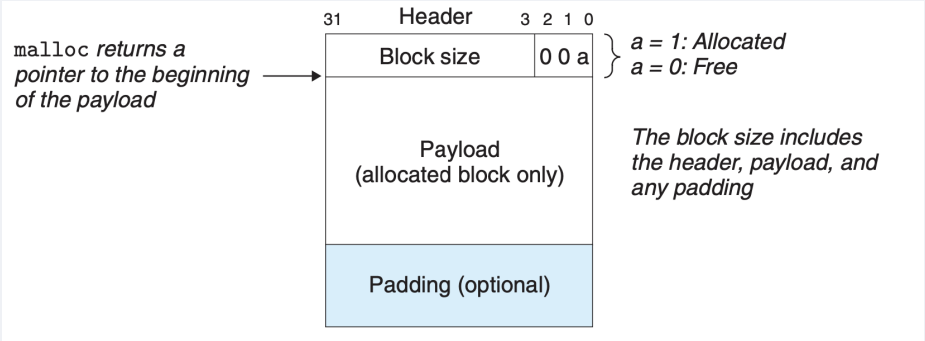
# Нюансы реализации

Если мы хотим добиться баланса между целями, нужно ответить на следующие вопросы:

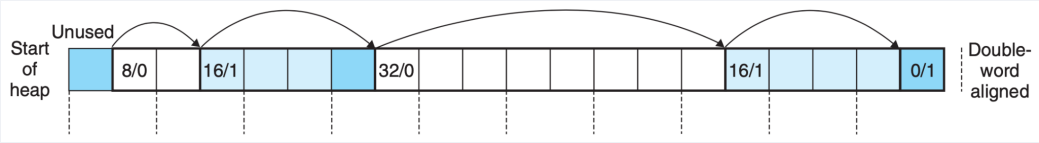
- **Free block organization** — как отслеживать свободные блоки?
- **Placement** — как выбрать свободный блок, куда мы будем аллоцировать?
- **Splitting** — как только заняли свободный блок, что делать с оставшейся частью?
- **Coalescing** — что делать с только что освобожденным блоком?

# Implicit Free List

Как различать границы блоков и понимать, свободен блок или нет?
Будем хранить все нужное в самом блоке:



# Как выглядит куча? (Free block organization)




- Последовательность занятых и свободных блоков
  - Односвязный список свободных блоков
- Время на операцию — линейное от количества **всех** блоков
  - Это недостаток, с которым в будущем будем бороться
- Выравнивание
  - Появляется минимальный размер блока — 2 слова

# Размещение выделенного блока (placement) I/II

При запросе аллокации — поиск подходящего свободного блока.
А как искать? Есть разные **политики размещения**!

## first fit — Берем первый попавшийся подходящего размера

- Обычно свободные блоки побольше оказываются в конце списка
- В начале свободные блоки меньше ⇒ поиск блока побольше займет больше времени

## next fit — Начинаем поиск там, где закончился предыдущий

- Есть вероятность, что следующий подходящий блок — остаток предыдущего
- Может работать быстрее, чем first fit
- Хуже утилизирует память

# Размещение выделенного блока (placement) II/II

При запросе аллокации — поиск подходящего свободного блока.
А как искать? Есть разные **политики размещения**!

**best fit — Перебираем все блоки и ищем подходящий с наименьшим размером**

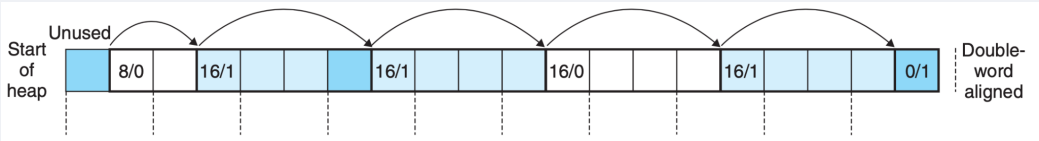
- Лучше утилизирует память
- Хуже по времени — бежим по всей куче

# Разделение свободных блоков (splitting)

Нашли подходящий свободный блок — сколько надо занять?

- Можем весь блок — плохо. Внутренняя фрагментация
- Можем делить на 2 части — выделенный блок и остаток

На примере — запрос на 3 слова:



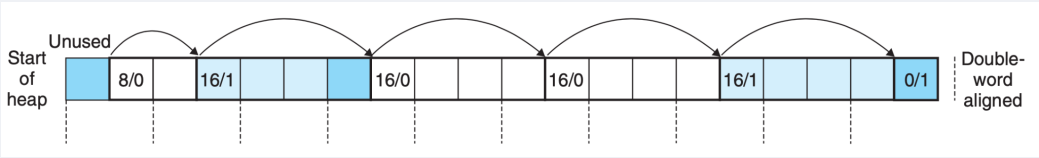
# Получение дополнительной памяти в куче

Что, если так и не нашли подходящий свободный блок?

1. Объединение свободных блоков (об этом дальше)
2. Если не сработало — просим у ОС больше памяти через sbrk
3. Превращаем новый кусок памяти в большой свободный блок
4. Вставляем блок в список и используем его для выделения

# Объединение свободных блоков (coalescing)

При освобождении свободные блоки могут оказаться рядом:




- Получили **ложную фрагментацию:**
  - Запрос на 4 слова не выполнится, хотя место есть
  - У нас два блока с payload = 3 слова
- Блоки надо объединять

# Нюансы объединения

Когда это можно делать?

## Сразу при запросе

- Быстро
- Может привести к лишним действиям

## Когда-то позже

- Если не нашли свободный блок, например
- Требует отдельного прохода по куче

# Нюансы объединения

Допустим, освободили текущий блок

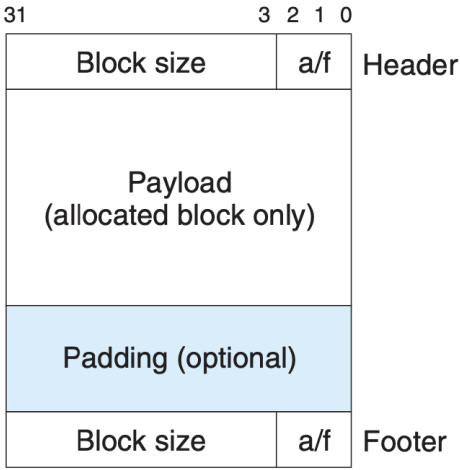
- Объединиться со следующим — легко за константное время
- А как объединяться с предыдущим?
- С текущей реализацией — только за линейное время от количества
- Получаем довольно медленный free

Как добиться константы?

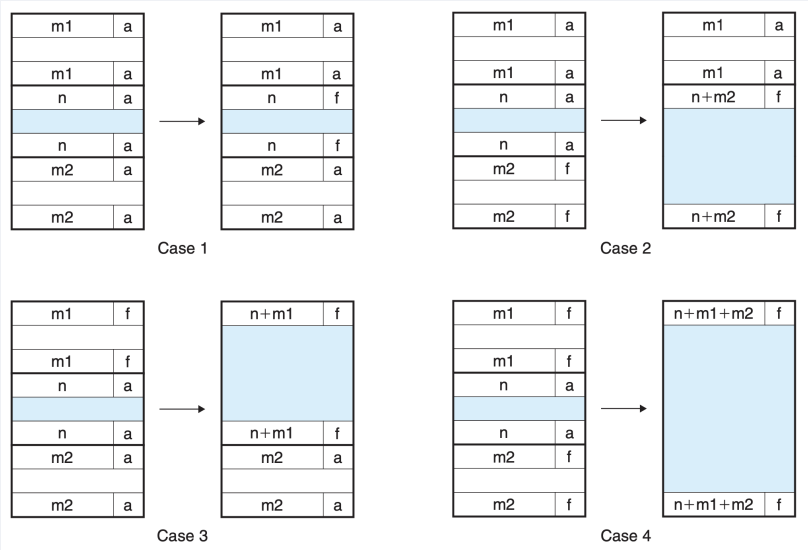
**Нам поможет Дональд Кнут!**

# Метод граничных маркеров

Улучшим наш блок:



# Метод граничных маркеров — случаи



# Метод граничных маркеров

- Получили константное время в каждом случае
- Подход легко обобщить на разные типы аллокаторов
- Тратим много памяти на header и footer
- Можем оптимизировать:
  - Можем избавиться от футера у выделенных блоков

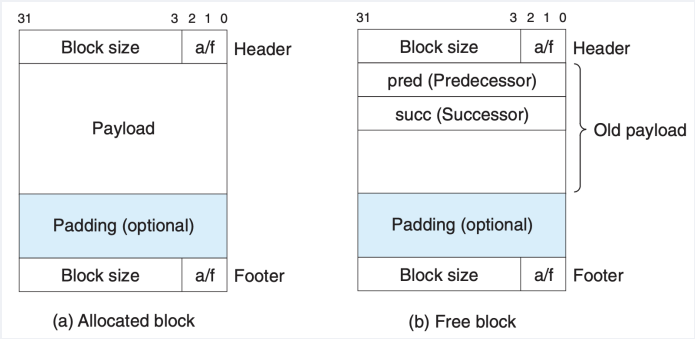
# Implicit Free Lists — что получили?

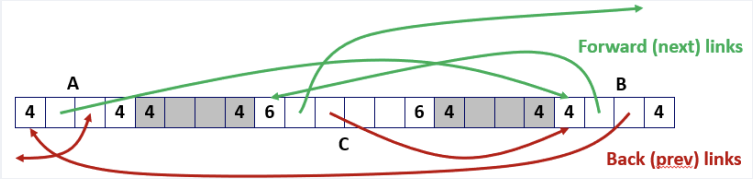
- Линейное время от количества всех блоков на аллокацию
- Константное время на освобождение
- Очень просто реализовать
- Редко используется из-за скорости malloc, но при этом довольно в определенных случаях может подойти

Разделение и объединение может распространяться почти на все аллокаторы!

# Explicit Free List

Строим **двусвязный список** из свободных блоков!



# Explicit Free List — аллокация блока

Также, как в Implicit Free List
Опять же, политики размещения бывают разные!

# Explicit Free List — освобождение блока

Зависит от **политики вставки в список**

## Last In First Out (LIFO) — Вставляем новый блок в начало

- free работает за константу

## Address order — Блоки в списке упорядочены по адресам

- free работает за линию — проходим по списку
- trade-off: мы лучше утилизируем память: проход first fit приближается к best fit!

Граничные маркеры все еще нужны для объединения блоков

# Explicit Free List — что получили?

По сравнению с Implicit Free List:

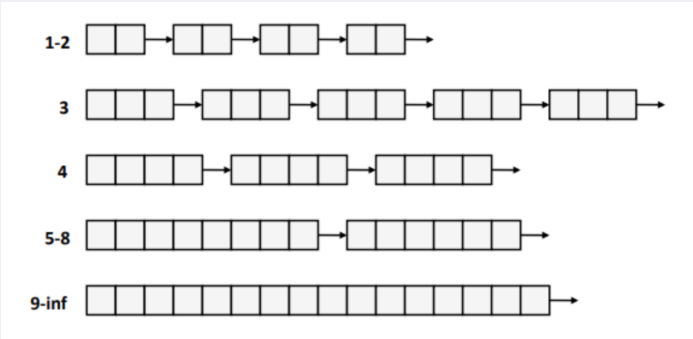
- Улучшили время аллокации: линейное время от количества свободных блоков, а не от количества всех блоков
- Нужно больше места на свободный блок, ведь мы храним указатели
- Это может увеличивать внутреннюю фрагментацию!

Время на аллокацию можно еще улучшить!

# Segregated Free Lists

Массив списков со свободными блоками определенных размеров

- Множество блоков разбиваем на классы по размерам
- Политик того, как разбить блоки на классы — много
  - Можно по степеням двойки
  - Можно маленькие размеры выделять в отдельные классы



# Segregated Free Lists — аллокация и освобождение

**Аллокация блока размера n:**

1. Ищем в списке свободных блоков подходящего класса
2. Если нашли свободный блок — выделяем
3. Разделить, а остаток поместить в нужный список — опционально
4. Если не нашли, пробуем искать в списке блоков большего размера
5. Перебрали все списки и не нашли — что делать?
   - Просим больше памяти у ОС при помощи `sbrk()`
   - Выделяем из новой памяти блок нужного размера
   - Остаток выделяем в отдельный блок и вставляем в нужный список

**Освобождение блока:**

- Вставка блока в нужный список
- Можем объединять блоки со вставкой в нужный список — опционально

# Segregated Free Lists — что мы уже получили?

- Время на запрос стало ниже
- Теперь исследуем не всю кучу, а какую-то часть
- Лучше используем память
- Проход с политикой first fit приближает best fit

То, что мы сейчас описали — концепция. Рассмотрим более конкретные реализации.

# Simple Segregated Storage

Каждый список хранит блоки одного размера
Пример: размеры {17-32} — округляем до 32

**Выделение блока:**

- Смотрим нужный список. Если не пуст — берем первый блок. Не разделяем.
- Список пуст — запрашиваем у ОС кусок памяти, и делим его на блоки нужного размера, теперь список не пуст.

**Освобождение:**

- Вставляем новый свободный блок в начало нужного списка
- Никакого объединения

# Simple Segregated Storage — что получили?

- `malloc` и `free` за константное время — круто!
- Уменьшили минимальный размер блока
  - Нам нужен только указатель на следующий блок
- Страдаем от внутренней фрагментации
  - Не разделяем же блоки
- Страдаем от внешней фрагментации
  - Есть конкретные сценарии
  - Много запросов на размер 1, много запросов на размер 2 ...

# Segregated Fits

- Каждый список — явный или неявный (как описывалось ранее)
  - В списке — блоки разных размеров!

**Выделение блока:**

1. Бежим по нужному списку по политике first fit
2. Нашли — делим, остаток отправляем в нужный список
3. Не нашли — ищем в списке класса больших размеров
4. Перебрали все списки? Просим памяти у ОС, выделяем нужный блок, остаток — помещаем в нужный список

**Освобождение блока:**

- Объединяем блоки и результат отправляем в нужный список

# Segregated Fits — что получили?

- Поиск не по всей куче, а по ее части
- first fit здесь приближается к best fit по всей куче
- **Популярный подход**
  - Используется в пакете malloc стандартной библиотеки Си

# Двоичные близнецы

- Segregated Fits, только каждый класс — степень двойки
  - Округляем размеры
- Пусть в куче $2^m$ слов
- Держим список для блоков размеров $2^k$, $0 \leq k \leq m$
- Изначально у нас один блок размером $2^m$

# Двоичные близнецы — выделение блока

Допустим, хотим выделить блок размером $2^k$

1. Надо найти первый доступный блок размером $2^j$, $k \leq j \leq m$
2. Рекурсивно:
   - Если $j = k$ — закончили
   - Если нет — рекурсивно делим блок пополам, пока $j$ не станет равным $k$
3. При делении оставшуюся часть (близнеца) отправляем в нужный список

Пример — минимальный размер блока — 64 K, запросили 34 K

| Step | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K | 64 K |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $2^4$ | | | | | | | | | | | | | | | |
| 2.1 | $2^3$ | | | | | | | | $2^3$ | | | | | | | |
| 2.2 | $2^2$ | | | | $2^2$ | | | | $2^3$ | | | | | | | |
| 2.3 | $2^1$ | | $2^1$ | | $2^2$ | | | | $2^3$ | | | | | | | |
| 2.4 | $2^0$ | $2^0$ | $2^1$ | | $2^2$ | | | | $2^3$ | | | | | | | |
| 2.5 | A: $2^0$ | $2^0$ | $2^1$ | | $2^2$ | | | | $2^3$ | | | | | | | |

# Двоичные близнецы — освобождение блока

Объединяем блоки, пока не дойдем до близнеца
Пример: освобождаем блок D

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 | A: $2^0$ | C: $2^0$ | $2^1$ | D: $2^1$ | $2^1$ | $2^3$ |
| 7.1 | A: $2^0$ | C: $2^0$ | $2^1$ | $2^1$ | $2^1$ | $2^3$ |
| 7.2 | A: $2^0$ | C: $2^0$ | $2^1$ | $2^2$ | | $2^3$ |

# Двоичные близнецы — что получили?

- Быстрый поиск и объединение
- Знаем адрес и размер блока — легко посчитать адрес близнеца
  - `xxx...x00000` — адрес блока размером 32 байта
  - `xxx...x10000` — адрес близнеца
  - Отличие в одном бите!
- Страдаем от внутренней фрагментации
- Может подойти, когда размеры выделенных блоков известны и они близки к степеням двойки

# Итог

Разных аллокаторов много — и они нужны:

- Разным вариантам ОС
- СУБД
- Реализациям языков программирования
- Разным программам на Си

Здесь рассмотрена только малая часть, их гораздо больше

# Благодарности

Рассказ про виртуальную память основан на материалах Я.А. Кириленко
Рассказ про аллокаторы — перевёрстан в LaTeX из презентации В.А. Кутуева