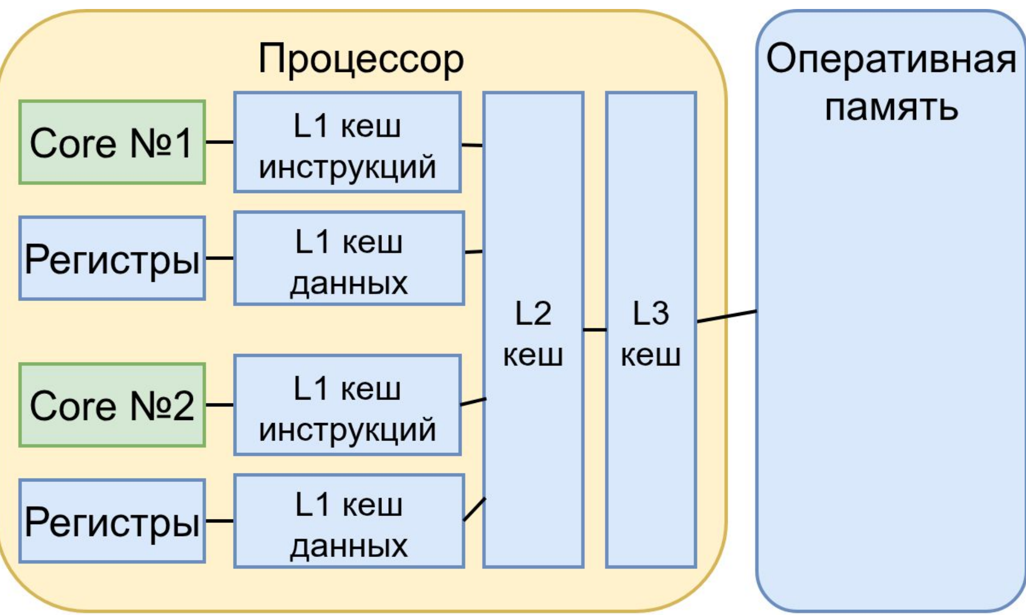
Управление кучей

Лекция №8

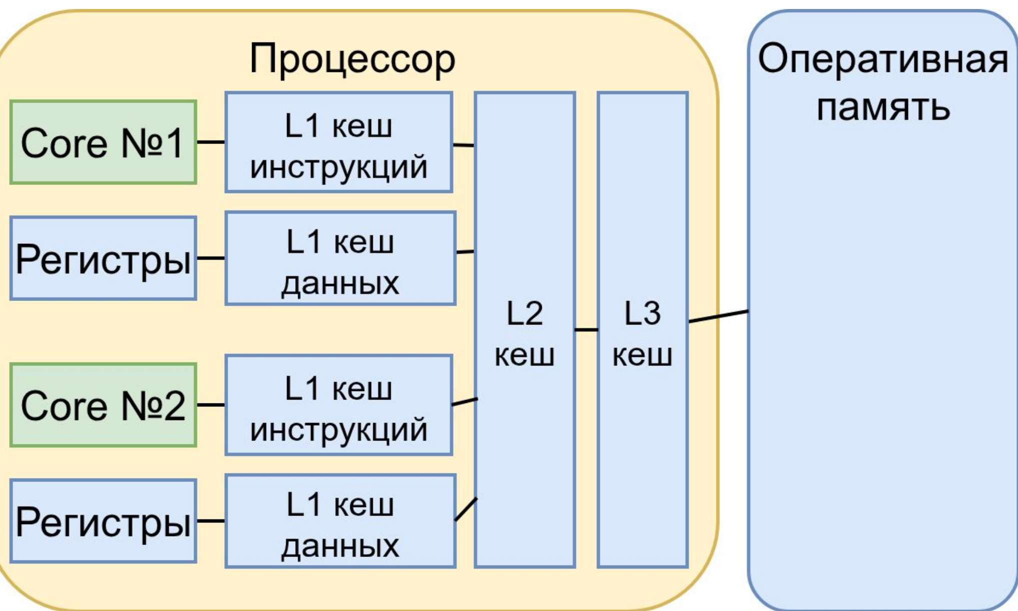
Теория языков программирования

Особенности памяти

Уровни памяти для процессоров AMD / Intel



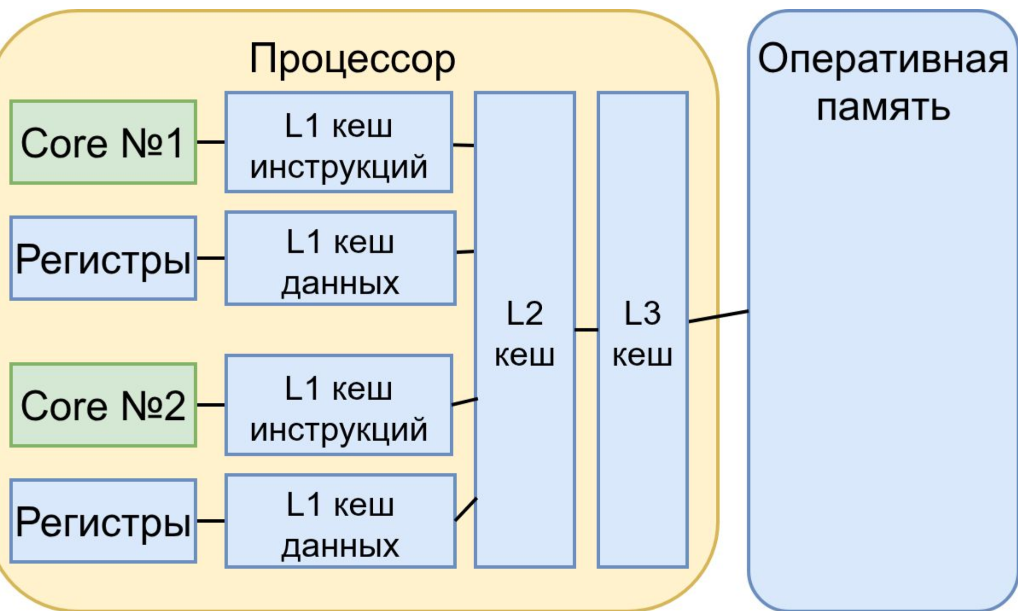
Уровни памяти для процессоров AMD / Intel



Особенности:

1. Память до 60-100 раз медленнее регистров
    - Это оптимизируется кэшем

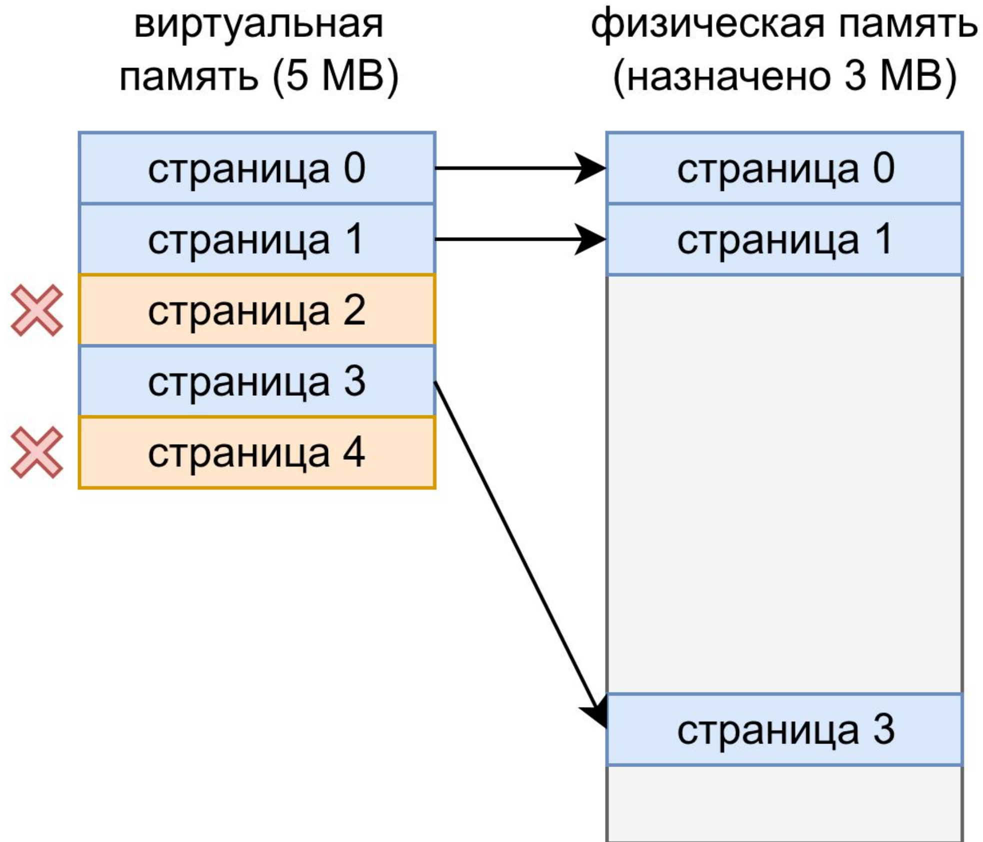
Уровни памяти для процессоров AMD / Intel



Особенности:

1. Память до 60-100 раз медленнее регистров
   - Это оптимизируется кэшем
2. Обращение к памяти почти всегда косвенное
   - Инструкции обращаются к регистрам
   - В регистрах могут лежать адреса в памяти

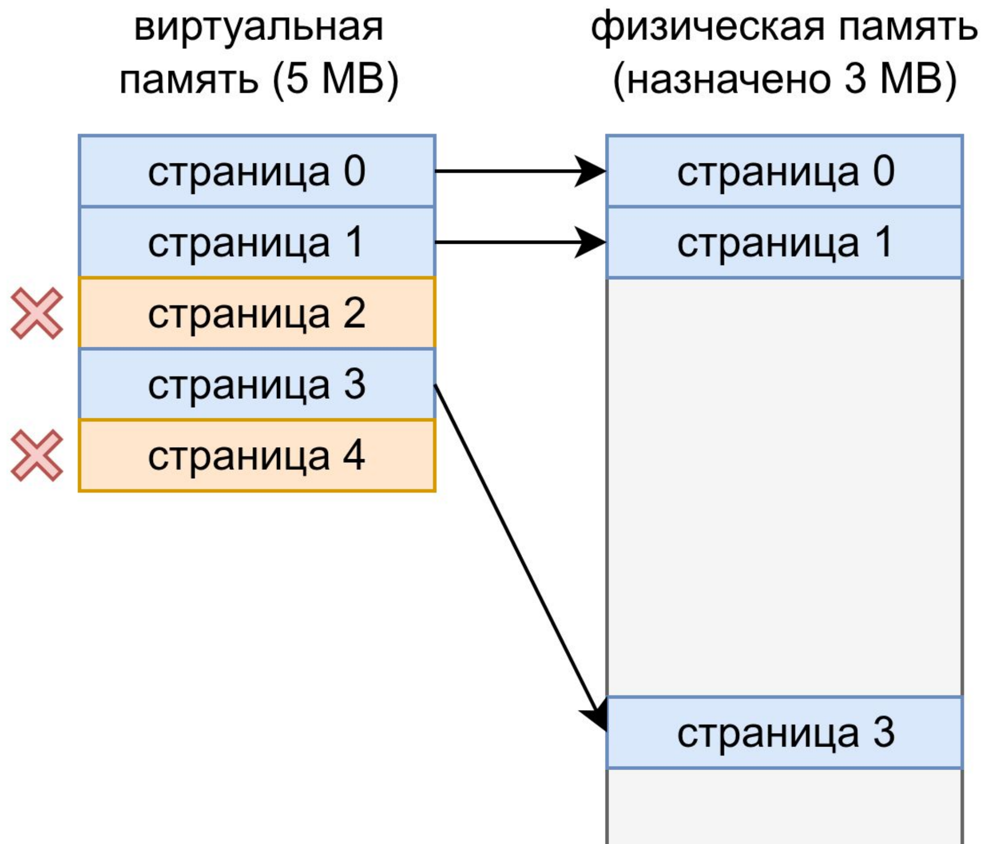
Виртуальная память



Overcommit в Linux:

1. Процесс запрашивает 5 фрагментов по 1 МБ
2. ОС выдаёт **виртуальные** адреса
3. **Физическая** память назначается, когда процесс начинает чтение/запись в страницу (4КБ)

Виртуальная память



- **VSS** (virtual set size) — объём виртуальной памяти
- **RSS** (resident set size) — объём физической памяти

NI	VIRT▽	RES	SHR	S	CPU%	MEM%
0	1408G	285M	121M	S	0.0	1.9
0	1408G	262M	112M	S	0.0	1.7
0	1403G	499M	187M	S	0.0	3.3
0	1394G	290M	147M	S	1.2	1.9
0	1394G	255M	98.0M	S	0.0	1.7

вывод htop в Linux

Куча (heap) в памяти

Сегменты памяти в процессе

статическая память
глобальные константы
глобальные переменные

thread-local storage
thread_local переменные

стек (stack)
frame 0
frame 1
frame 2
frame 3
растёт вниз ↓

куча (heap)
object 4
освобождено
object 3
object 2
освобождено
object 1
растёт вверх ↑

Сегменты памяти в процессе

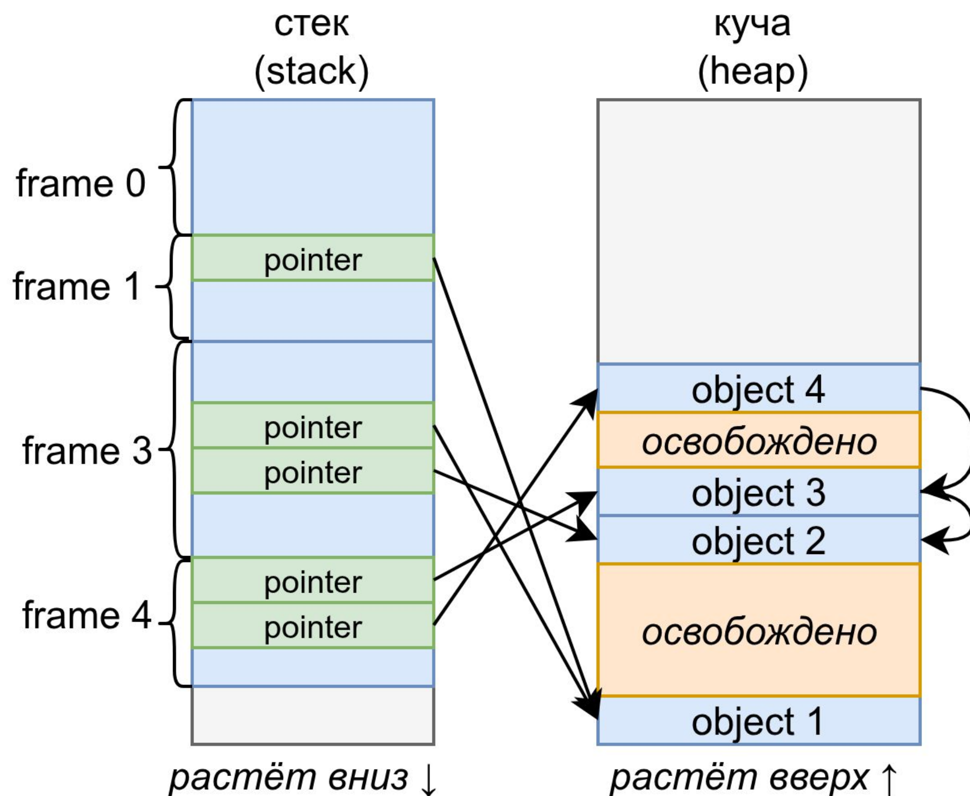
Имеют предсказуемые адреса:

- Статическая память
- TLS
- Стек

Адресуются косвенно:

- Куча

Упрощаем модель



Допустим, что в программе:

1. Стек хранит указатели
2. Куча хранит объекты
   - в полях объектов тоже есть указатели

Аллокаторы

Аллокаторы

Allocator выделяет память

- `void* malloc(size_t size)`
- `T* new(...)`

Аллокаторы

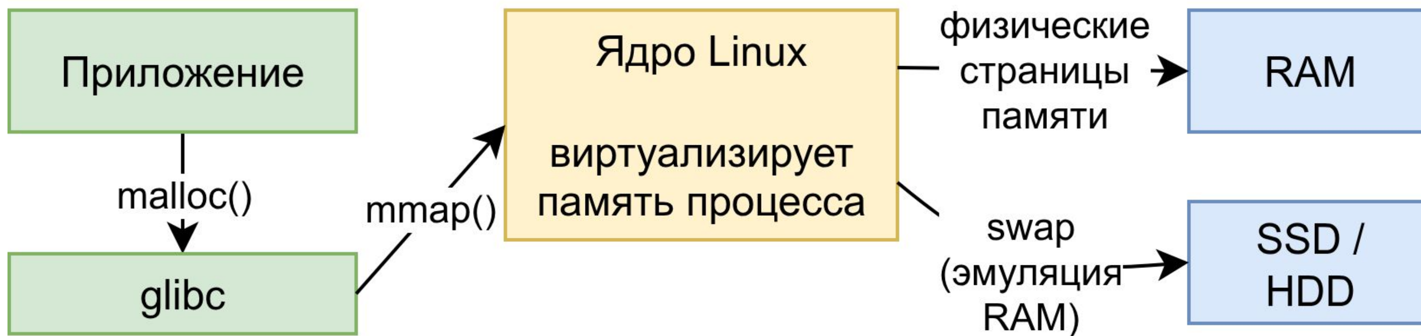
Allocator выделяет память

- `void* **malloc**(size_t size)`
- `T* **new**(...)`

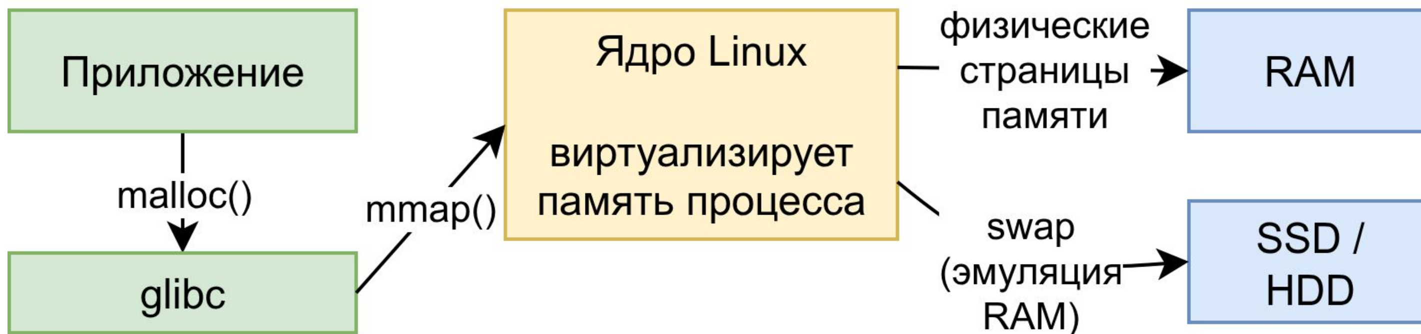
Allocator позволяет освободить память

- `void **free**(void* ptr)`
- `**delete**(T* ptr)`

Аллокатор glibc — ptmalloc2



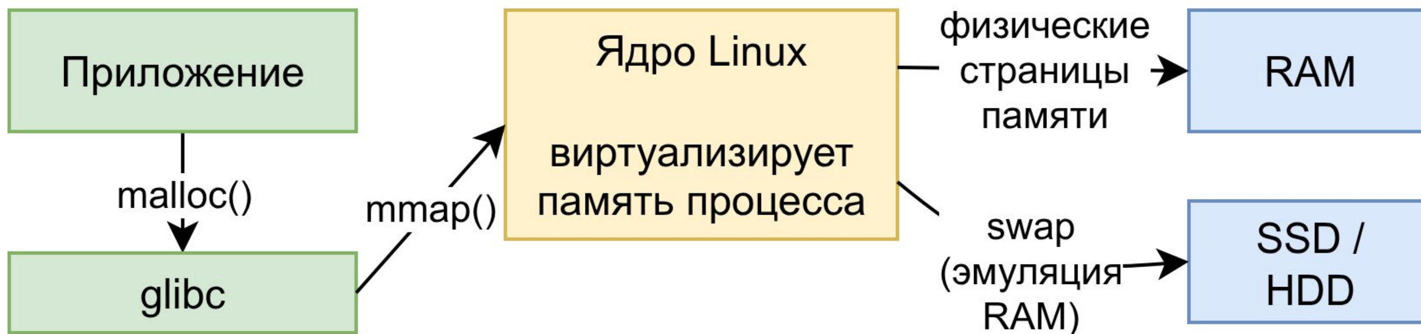
Аллокатор glibc — ptmalloc2



Дилемма аллокатора:

1. `malloc()` и `free()` должны быть быстрыми
2. надо беречь память для остальных процессов

Аллокатор glibc — ptmalloc2



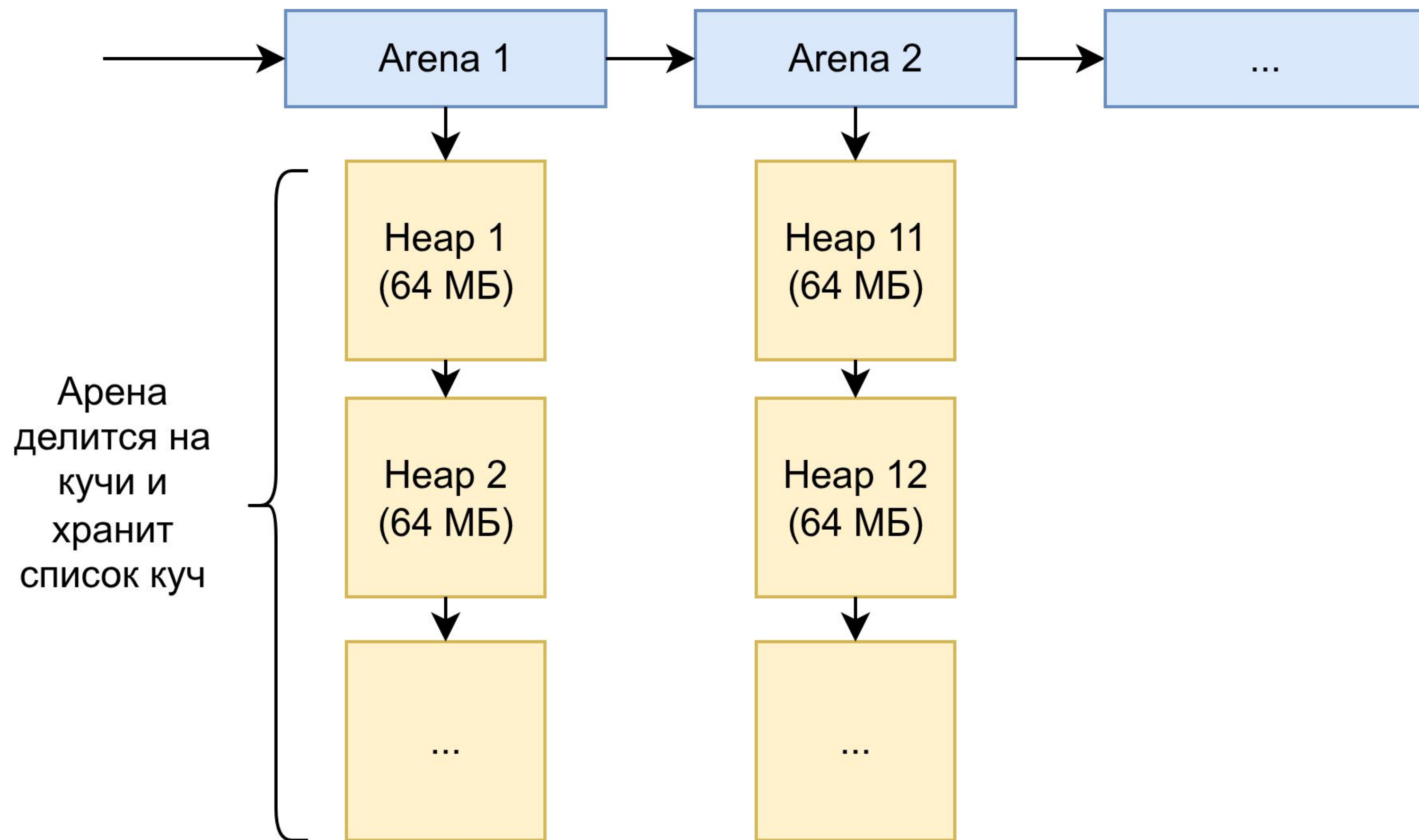
Дилемма аллокатора:

1. `malloc()` и `free()` должны быть быстрыми
2. надо беречь память для остальных процессов

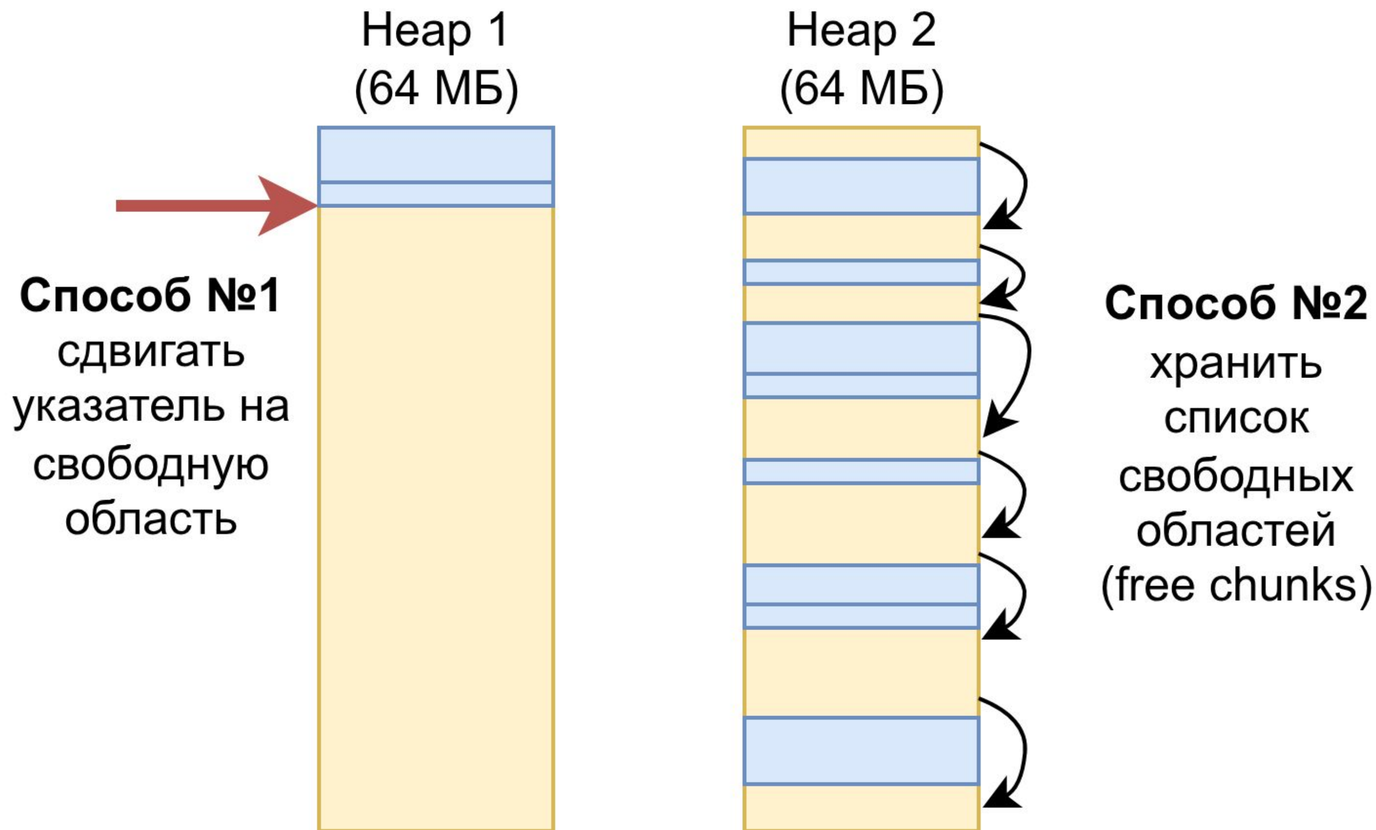
Решения:

1. выделять много памяти и полагаться на overcommit
2. хорошо переиспользовать память

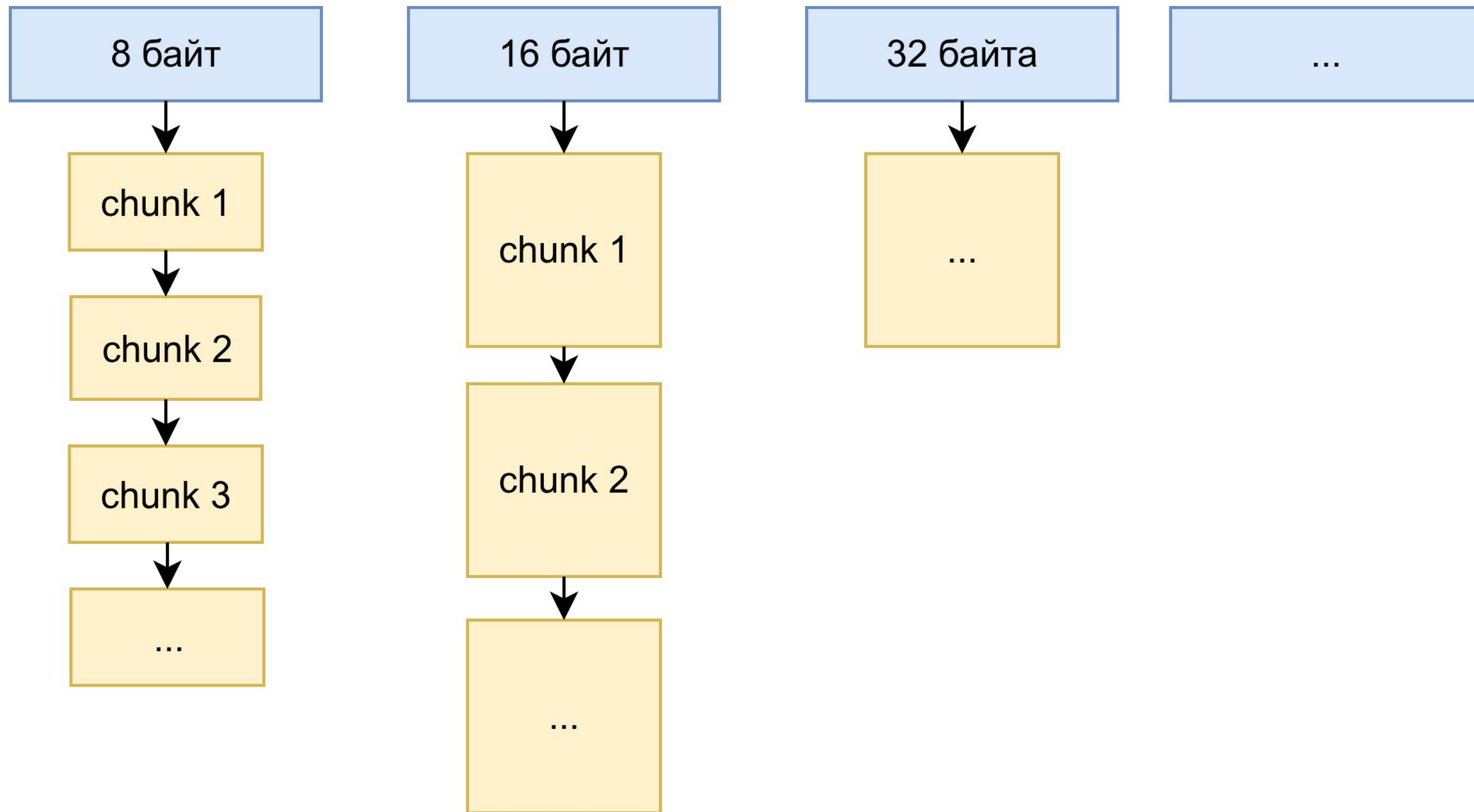
Арены и кучи



Деление кучи на области (chunks)



Деление free chunks по размерам



Высоконагруженные аллокаторы

Особенности оборудования

1. Локальность данных важна
    - Доступ к RAM — 60-100 тактов
    - Доступ к L1 кешу — 1-3 такта

Особенности оборудования

1. Локальность данных важна
    - Доступ к RAM — 60-100 тактов
    - Доступ к L1 кешу — 1-3 такта
    - Spatial locality — связанные данные находятся рядом
    - Temporal locality — к данным обращаются повторно, пока они в кеше

Особенности оборудования

1. Локальность данных важна
    - Доступ к RAM — 60-100 тактов
    - Доступ к L1 кешу — 1-3 такта
    - Spatial locality — связанные данные находятся рядом
    - Temporal locality — к данным обращаются повторно, пока они в кеше
2. Многопроцессорные системы
    - Конкуренция потоков — это плохо

Особенности оборудования

1. Локальность данных важна
    - Доступ к RAM — 60-100 тактов
    - Доступ к L1 кешу — 1-3 такта
    - Spatial locality — связанные данные находятся рядом
    - Temporal locality — к данным обращаются повторно, пока они в кеше
2. Многопроцессорные системы
    - Конкуренция потоков — это плохо
    - Блокировки (mutex) — большие накладные расходы

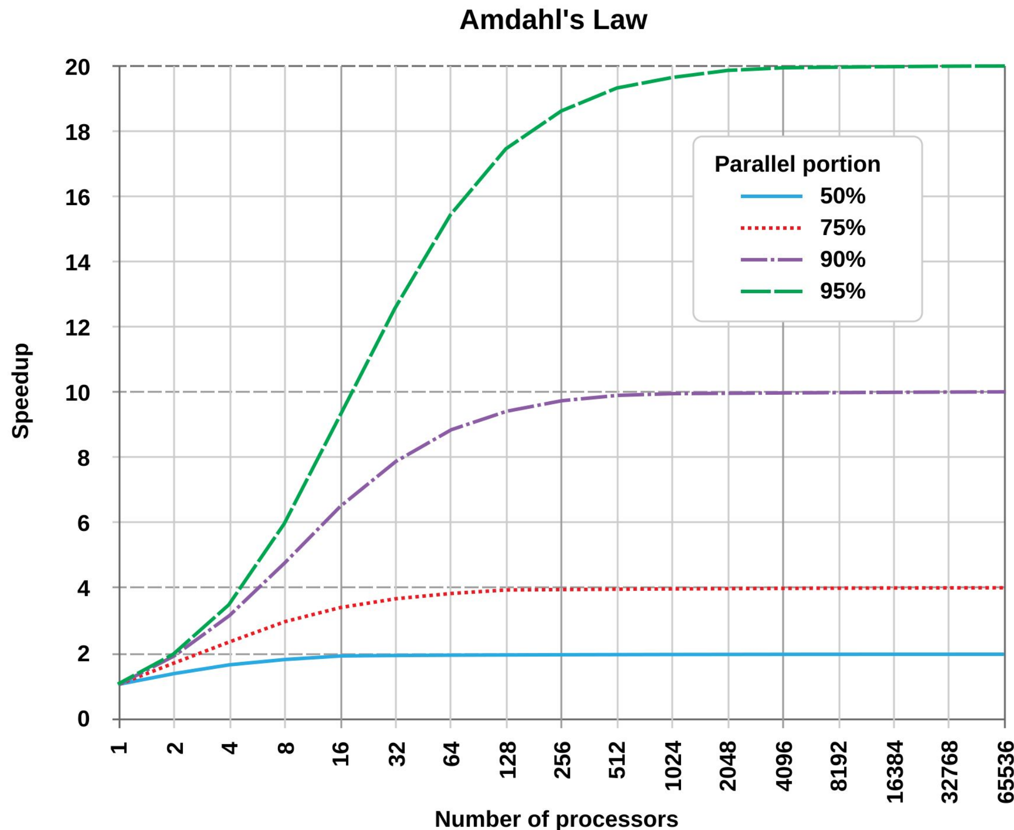
Особенности оборудования

1. Локальность данных важна
    - Доступ к RAM — 60-100 тактов
    - Доступ к L1 кешу — 1-3 такта
    - Spatial locality — связанные данные находятся рядом
    - Temporal locality — к данным обращаются повторно, пока они в кеше
2. Многопроцессорные системы
    - Конкуренция потоков — это плохо
    - Блокировки (mutex) — большие накладные расходы
    - Атомарный доступ (spinlock, lock-free) — накладные расходы

Особенности оборудования

1. Локальность данных важна
   - Доступ к RAM — 60-100 тактов
   - Доступ к L1 кешу — 1-3 такта
   - Spatial locality — связанные данные находятся рядом
   - Temporal locality — к данным обращаются повторно, пока они в кеше
2. Многопроцессорные системы
   - Конкуренция потоков — это плохо
   - Блокировки (mutex) — большие накладные расходы
   - Атомарный доступ (spinlock, lock-free) — накладные расходы
   - Кеш L1 — свой у каждого ядра, возможен false sharing

Закон Амдала



Допустим

- **p** — число процессоров
- **a** — доля операций, выполняемых последовательно

Ускорение параллелизма равно

$$1 / (a + (1 - a) / p)$$

Пусть a=0.4, p=16

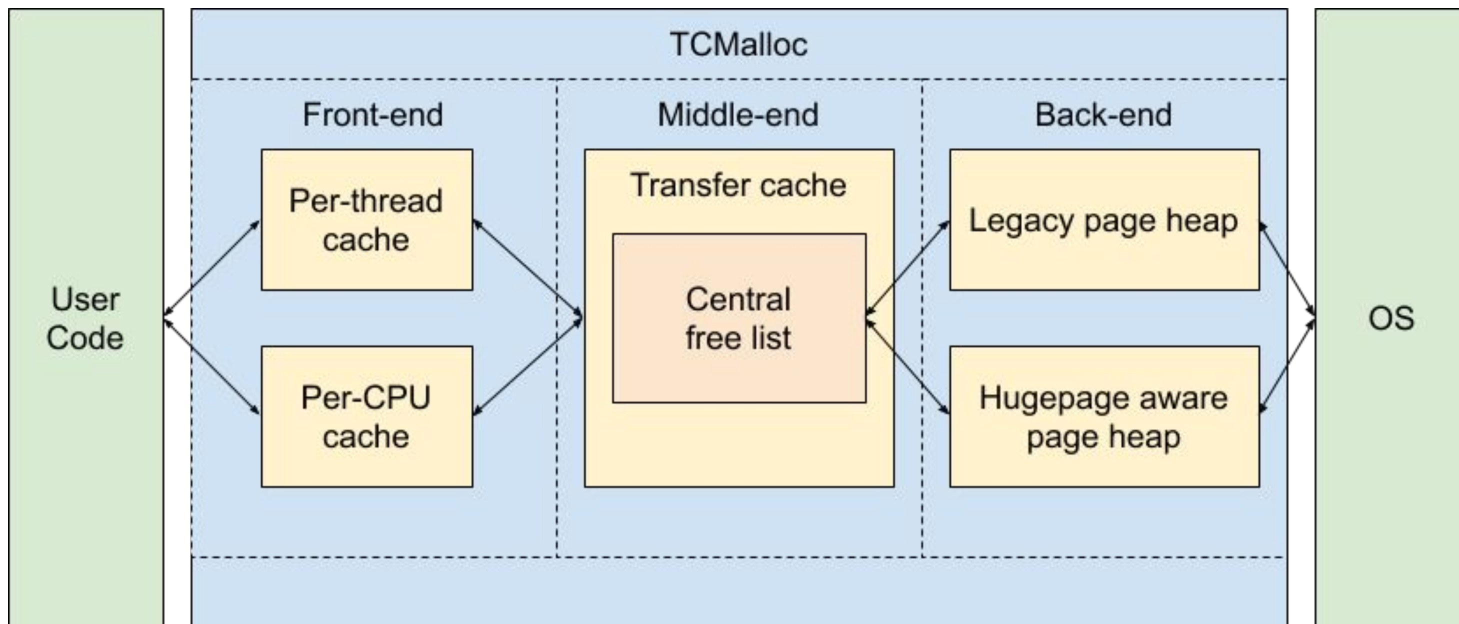
$$1 / (0.4 + (1 - 0.4) / 16)$$

$$\approx \text{в } 2.29 \text{ раз}$$

Amdahl's law - Wikipedia

Решение

Свой кеш аллокатора на поток или на ядро CPU
Где используется: tcmalloc, jemalloc



TCMalloc : Thread-Caching Malloc

Практический совет

Выигрывают структуры данных с хорошим spatial locality

Практический совет

Выигрывают структуры данных с хорошим spatial locality

Структуры «плохих» структур:

1. Связный список
2. Деревья, особенно с вращением (AVL)
3. Хеш с закрытой адресацией

Практический совет

Выигрывают структуры данных с хорошим spatial locality

Структуры «хороших» структур:

1. Массив и его производные
    - Flat Map — сортированный массив как словарь
    - Ring Buffer — кольцевой буфер для очередей и deque
    - Куча (heap) — дерево в массиве

Практический совет

Выигрывают структуры данных с хорошим spatial locality

Структуры «хороших» структур:

1. Массив и его производные
   - Flat Map — сортированный массив как словарь
   - Ring Buffer — кольцевой буфер для очередей и deque
   - Куча (heap) — дерево в массиве
2. Развёрнутый связный список (Unrolled Linked List)
3. Хеш с открытой адресацией

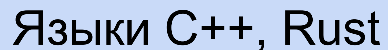
Освобождение памяти

Способы освобождения памяти программой

1. Ручное управление — `void free(void*)` и `delete(T*)`

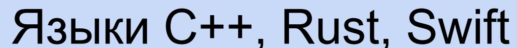
Способы освобождения памяти программой

1. Ручное управление — `void free(void*)` и `delete(T*)`
2. Семантика владения
    - `unique_ptr<T>`
    - move-семантика



Способы освобождения памяти программой

1. Ручное управление — `void free(void*)` и `delete(T*)`
2. Семантика владения
    - `unique_ptr<T>`
    - move-семантика
3. Подсчёт ссылок
    - strong pointer — `shared_ptr<T>`
    - weak pointer — `weak_ptr<T>`



Способы освобождения памяти программой

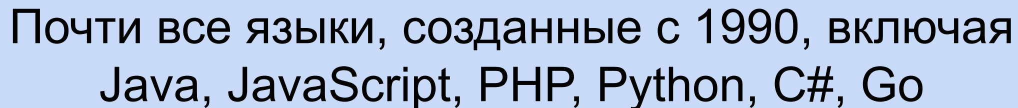
1. Ручное управление — `void free(void*)` и `delete(T*)`
2. Семантика владения
   - `unique_ptr<T>`
   - move-семантика
3. Подсчёт ссылок
   - strong pointer — `shared_ptr<T>`
   - weak pointer — `weak_ptr<T>`
4. Деревья объектов
   - объект имеет родителя
   - удаление родителя удаляет всё дерево

Способы освобождения памяти программой

1. Ручное управление — `void free(void*)` и `delete(T*)`
2. Семантика владения
    - `unique_ptr<T>`
    - move-семантика
3. Подсчёт ссылок
    - strong pointer — `shared_ptr<T>`
    - weak pointer — `weak_ptr<T>`
4. Деревья объектов
    - объект имеет родителя
    - удаление родителя удаляет всё дерево
5. Пул памяти (arena allocation)

Способы освобождения памяти программой

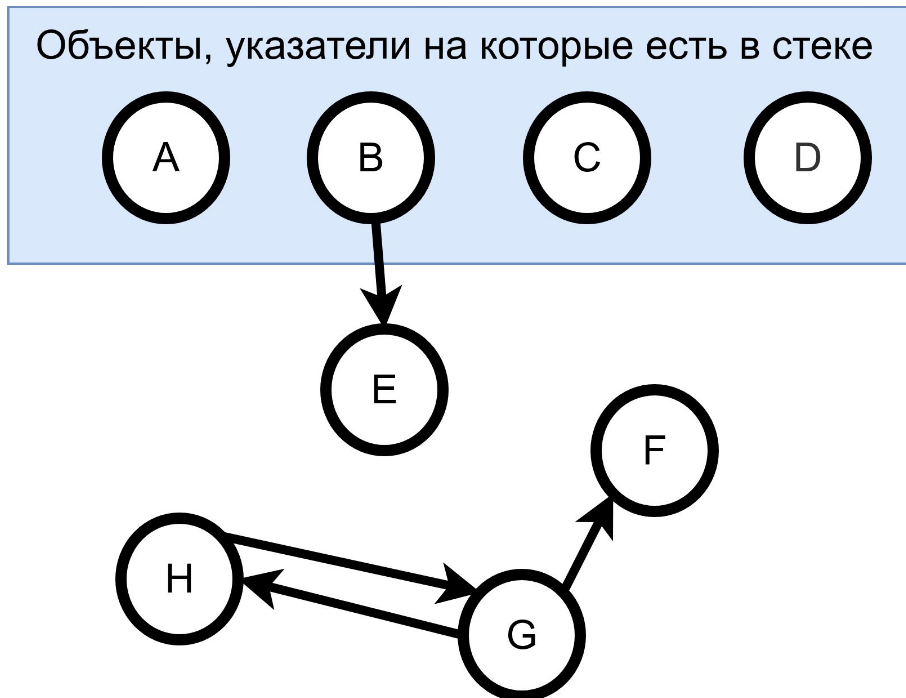
1. Ручное управление — `void free(void*)` и `delete(T*)`
2. Семантика владения
   - `unique_ptr<T>`
   - move-семантика
3. Подсчёт ссылок
   - strong pointer — `shared_ptr<T>`
   - weak pointer — `weak_ptr<T>`
4. Деревья объектов
   - объект имеет родителя
   - удаление родителя удаляет всё дерево
5. Пул памяти (arena allocation)
6. Сборка мусора



Сборка мусора
(Garbage Collection, GC)

Польза сборки мусора

1. Написание кода становится проще
2. GC может найти циклы объектов



Цена сборки мусора

1. Паузы из-за сборки
   - Полный Stop the world — если GC без конкуренции
   - Частичные остановки и блокировки — если GC конкурентный
2. Перерасход памяти либо CPU
   - Частый вызов GC — пустая трата CPU
   - Редкий вызов GC — мусор занимает память
3. Меняется баланс простого и сложного
   - Обычный код писать проще
   - Высокопроизводительный — сложнее

Влияние GC на дизайн языка

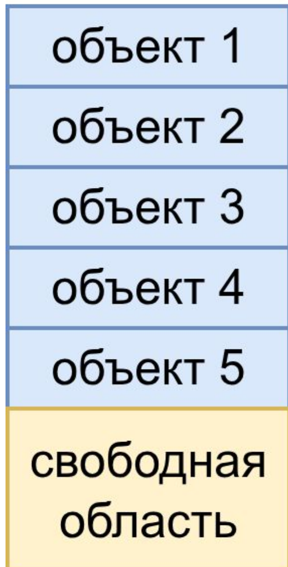
1. Адресная арифметика мешает GC
2. Запись указателей становится дороже
3. Нужна поддержка компилятора / интерпретатора

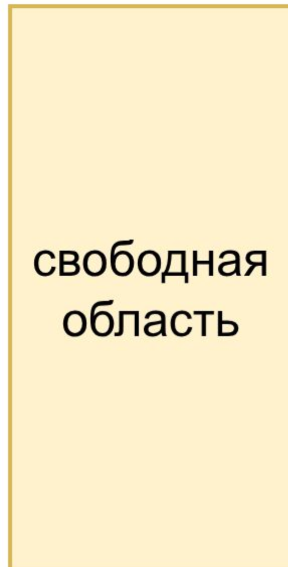
Типы GC по особенностям работы

1. Консервативные (conservative) и точные (precise)
2. Конкурентные (concurrent) и неконкурентные
3. Перемещающие (moving) и неперемещающие
4. Поколенческие GC (generational)

Общие алгоритмы GC

Алгоритм semi-space GC

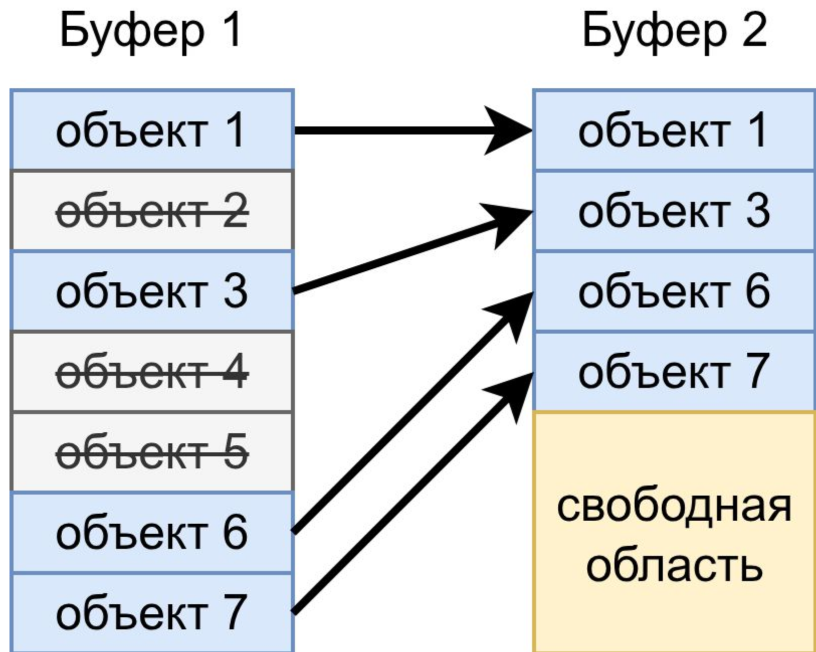




Использует два буфера:

1. Сначала заполняем один буфер

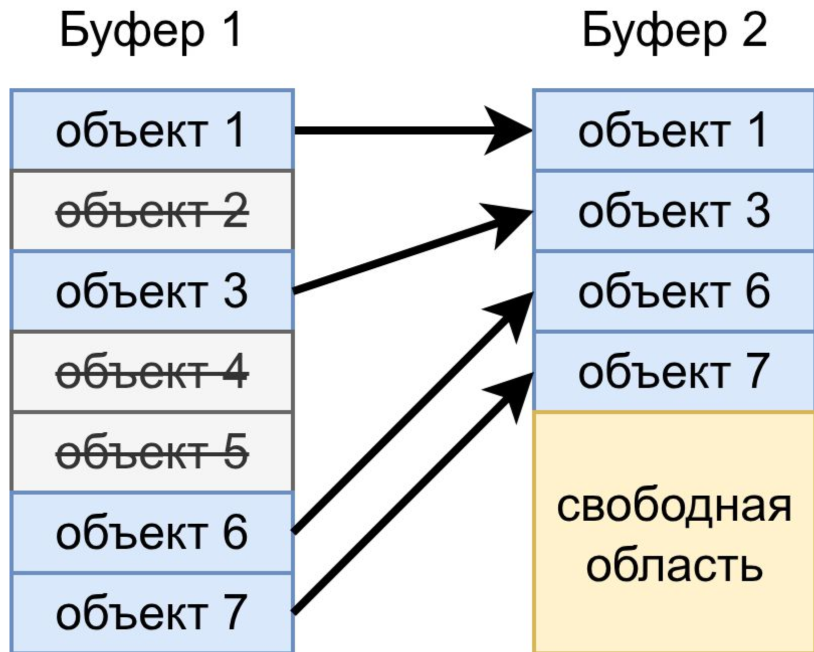
Алгоритм semi-space GC



Использует два буфера:

1. Сначала заполняем один буфер
2. Затем перемещаем буфер

Алгоритм semi-space GC



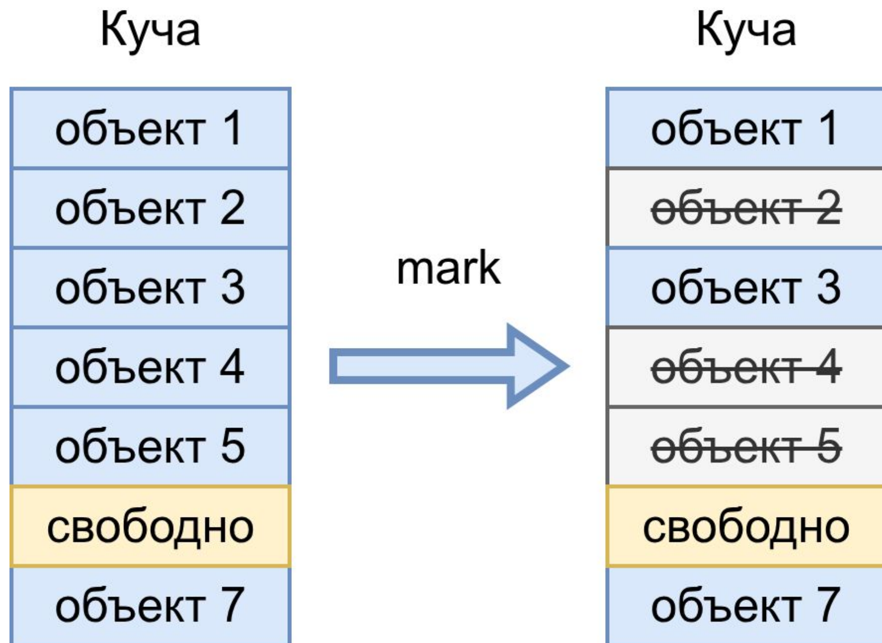
Использует два буфера:

1. Сначала заполняем один буфер
2. Затем перемещаем буфер

Semi-space GC не используется на практике из-за

- двойного расхода
- большого числа перемещений

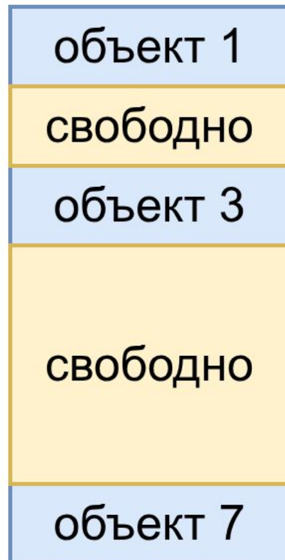
Алгоритм mark-and-sweep GC



Этап **mark**:

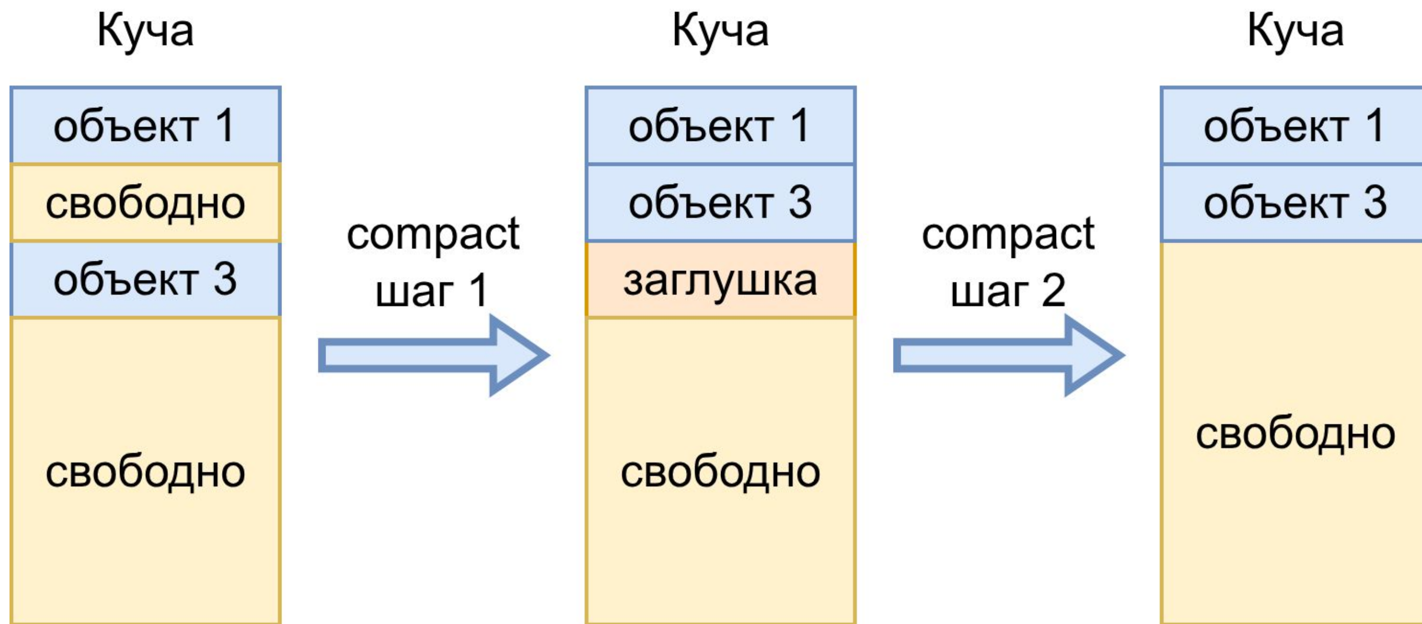
1. Сканируем стек, статическую память, TLS
2. Отмечаем недостижимые из них объекты

Алгоритм mark-and-sweep GC



Этап **sweep**: освобождаем недостижимые объекты

Алгоритм mark-and-sweep GC

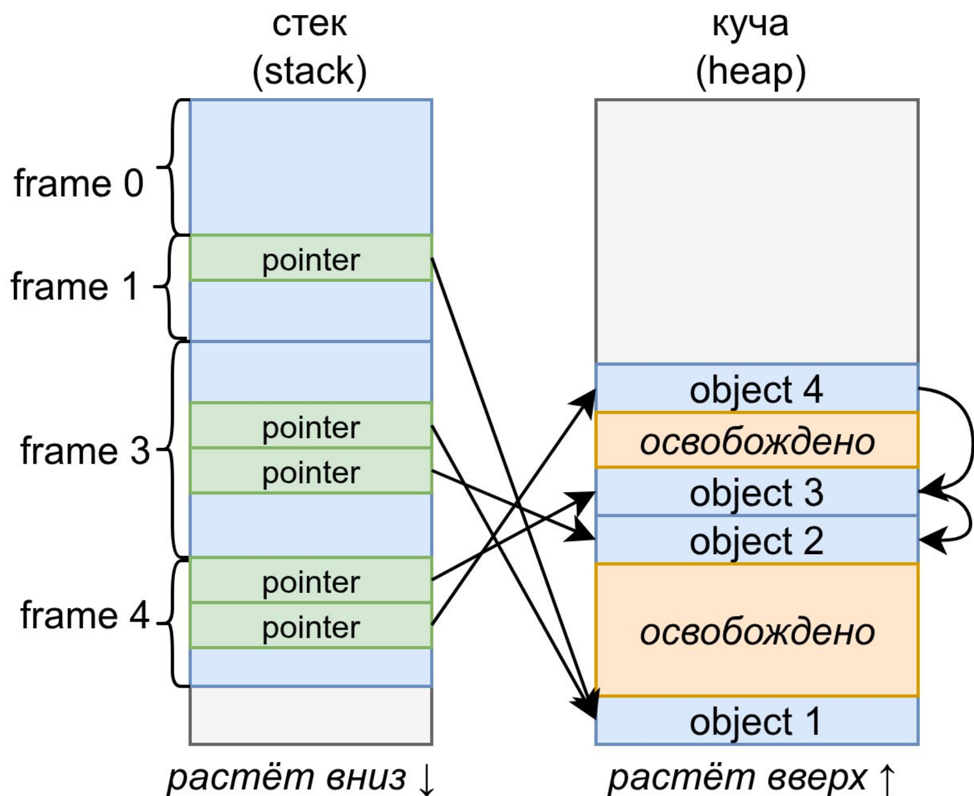


Этап **compact***: уплотняем расположение объектов в куче

* только для перемещающих GC

Как устроен этап Mark

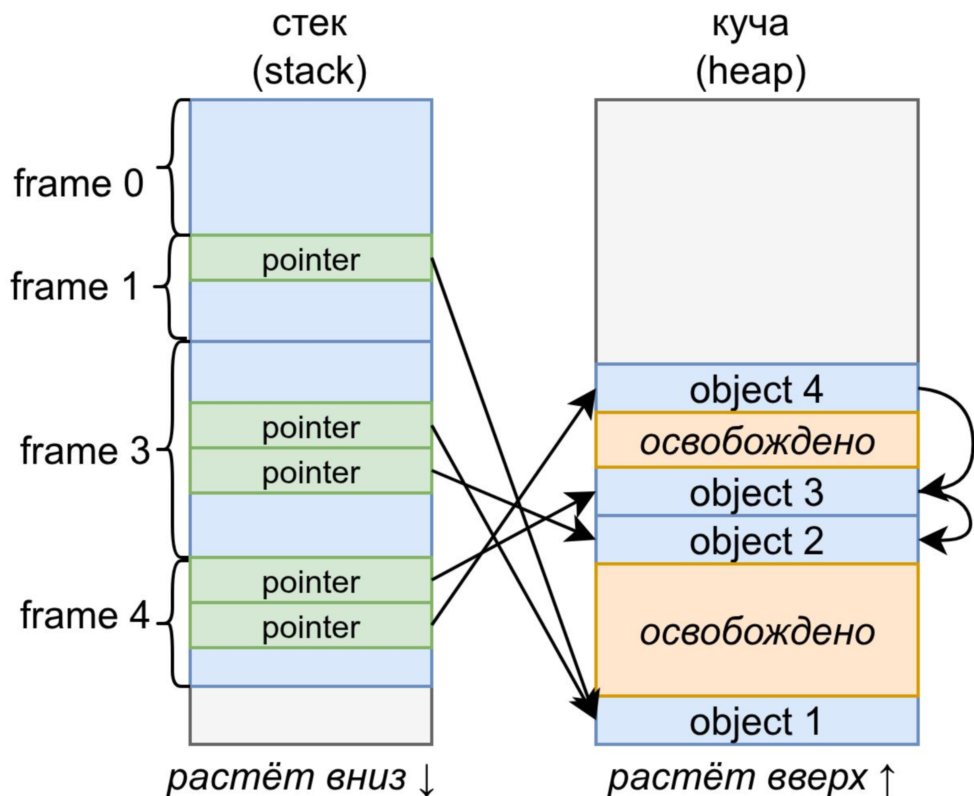
Мусор — это недостижимые объекты



Допустим, что в программе:

1. Стек хранит указатели
2. Куча хранит объекты
   - в полях объектов тоже есть указатели

Мусор — это недостижимые объекты



Допустим, что в программе:

1. Стек хранит указатели
2. Куча хранит объекты
   - в полях объектов тоже есть указатели

Можно:

1. просканировать стек и найти достижимые объекты
2. пройти по этим объектам и найти косвенно достижимые

Three-color garbage collector — обход графа в ширину

1. В начале все объекты **белые**

Three-color garbage collector — обход графа в ширину

1. В начале все объекты **белые**
2. Обходим стек, статическую память и TLS
    - Красим найденные объекты в **серый**

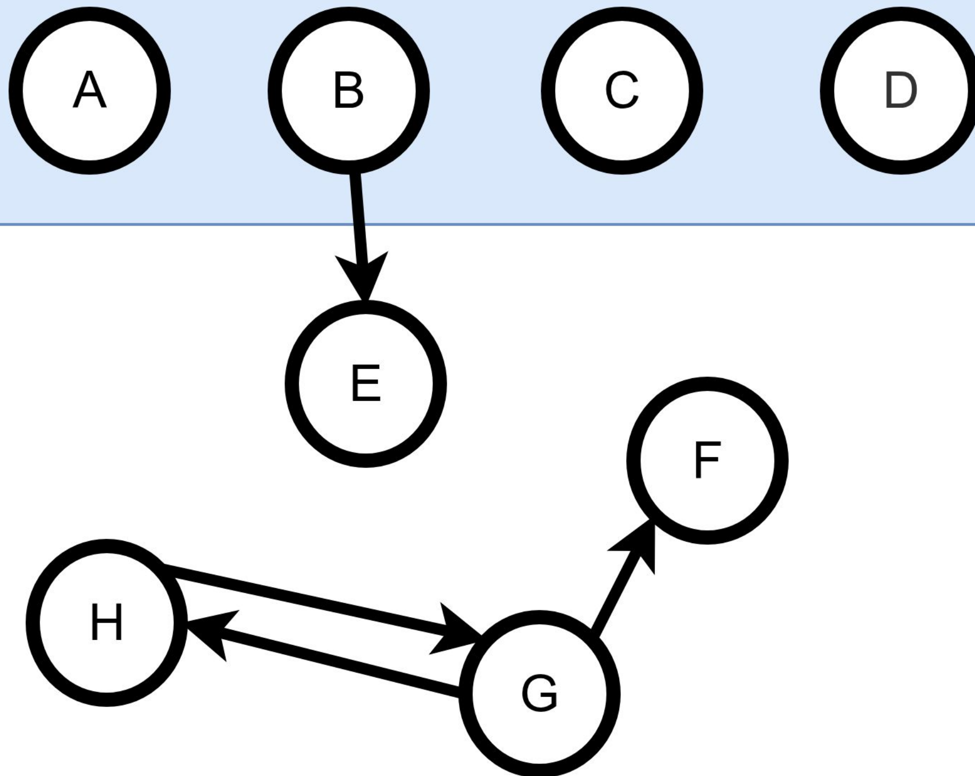
Three-color garbage collector — обход графа в ширину

1. В начале все объекты **белые**
2. Обходим стек, статическую память и TLS
   - Красим найденные объекты в **серый**
3. Обходим серые объекты, пока они не кончатся
   - Красим найденные объекты в **серый**
   - Красим пройденные объекты в **чёрный**

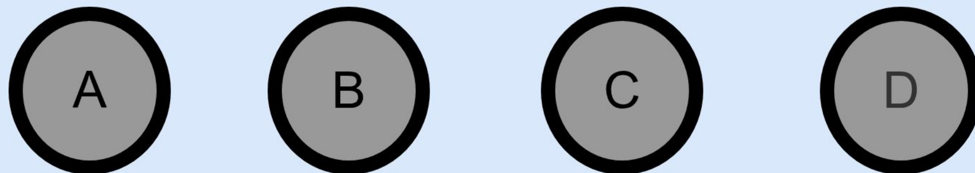
Three-color garbage collector — обход графа в ширину

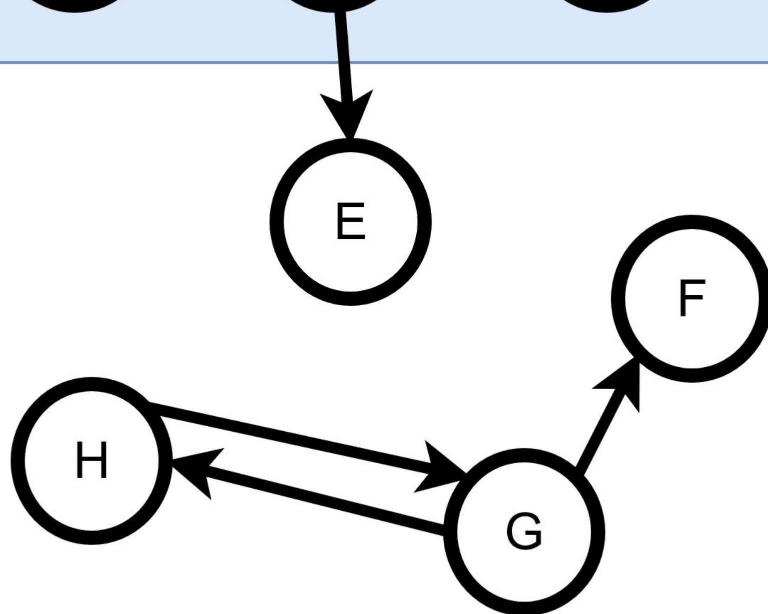
1. В начале все объекты **белые**
2. Обходим стек, статическую память и TLS
   - Красим найденные объекты в **серый**
3. Обходим серые объекты, пока они не кончатся
   - Красим найденные объекты в **серый**
   - Красим пройденные объекты в **чёрный**
4. Оставшиеся **белые** объекты — это мусор

Three-color garbage collector — обход графа в ширину

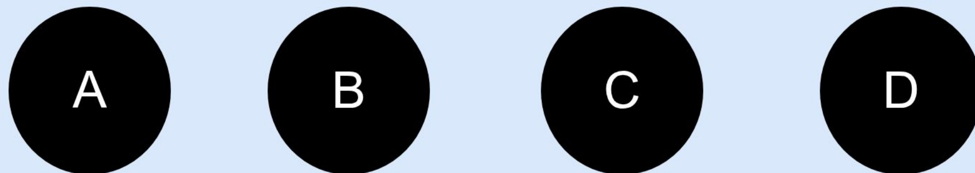


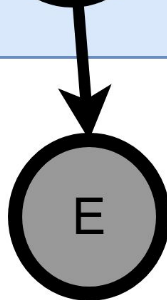
Three-color garbage collector — обход графа в ширину

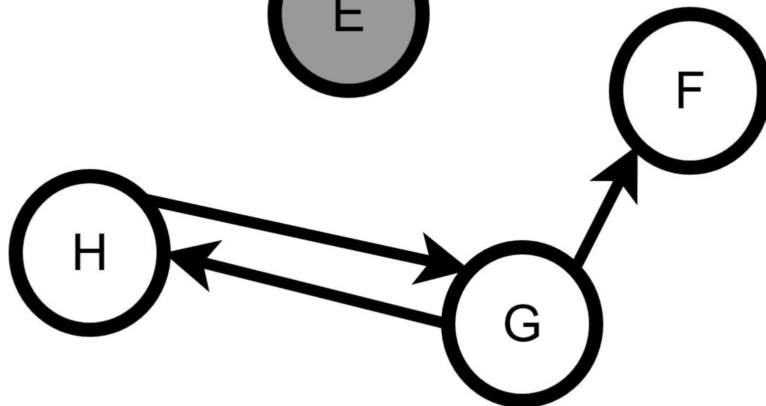




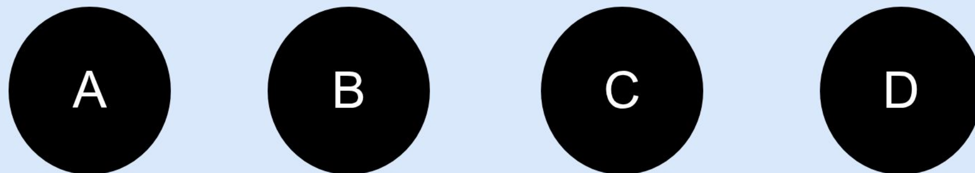
Three-color garbage collector — обход графа в ширину

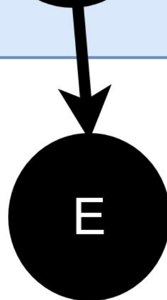


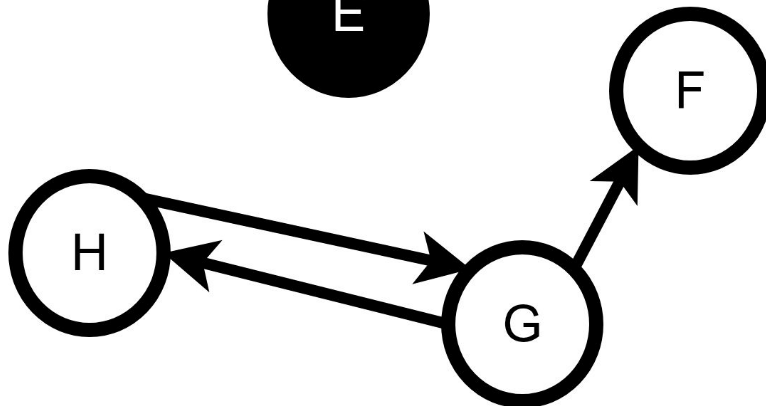




Three-color garbage collector — обход графа в ширину





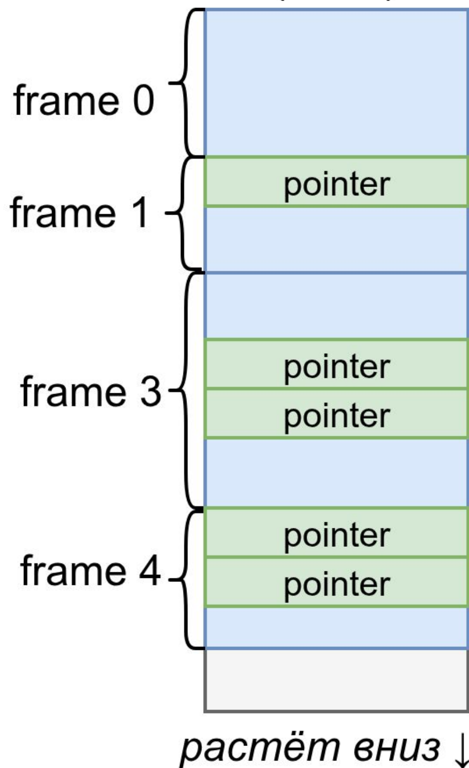


Сканирование стека

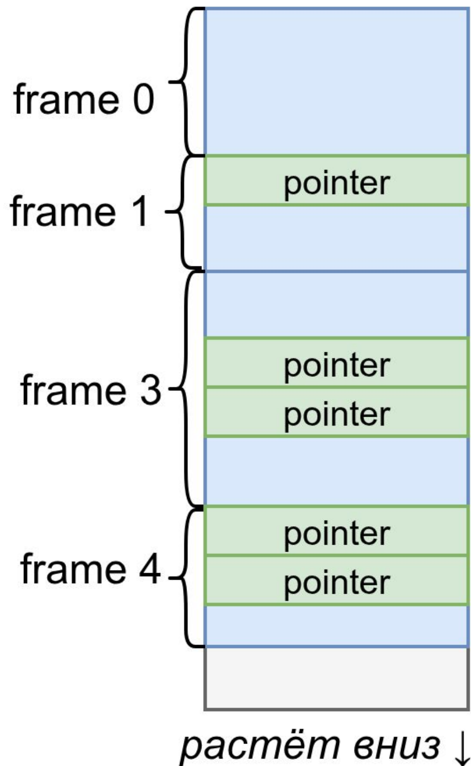
Память процесса в Linux на AMD64

1. Куча имеет фиксированный диапазон адресов
2. Огромное пространство между стеком и кучей
   - ≈ 2^{48} байт
   - это виртуальная память

Как на стеке найти указатели на кучу?



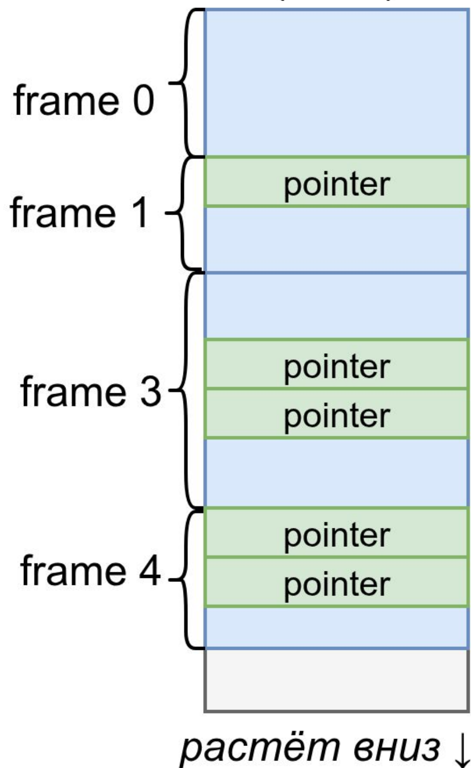
Как на стеке найти указатели на кучу?



Соображения для AMD64:

1. Указатели выровнены по границе 8 байт

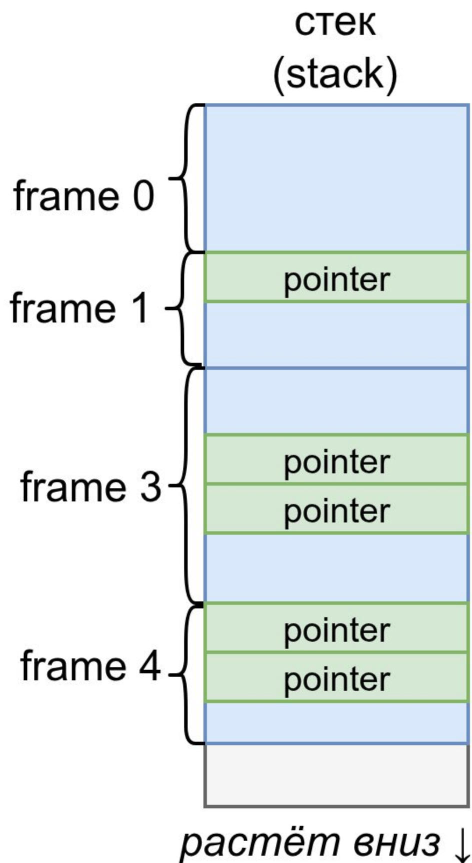
Как на стеке найти указатели на кучу?



Соображения для AMD64:

1. Указатели выровнены по границе 8 байт
2. Куча имеет фиксированный диапазон адресов

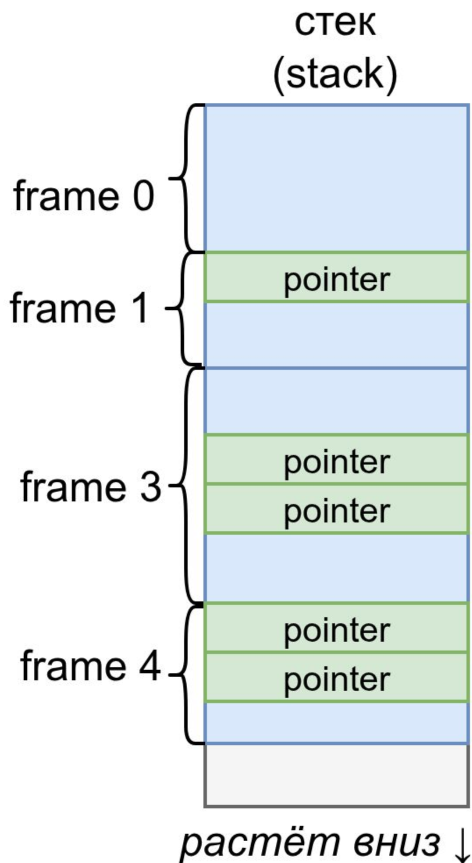
Как на стеке найти указатели на кучу?



Соображения для AMD64:

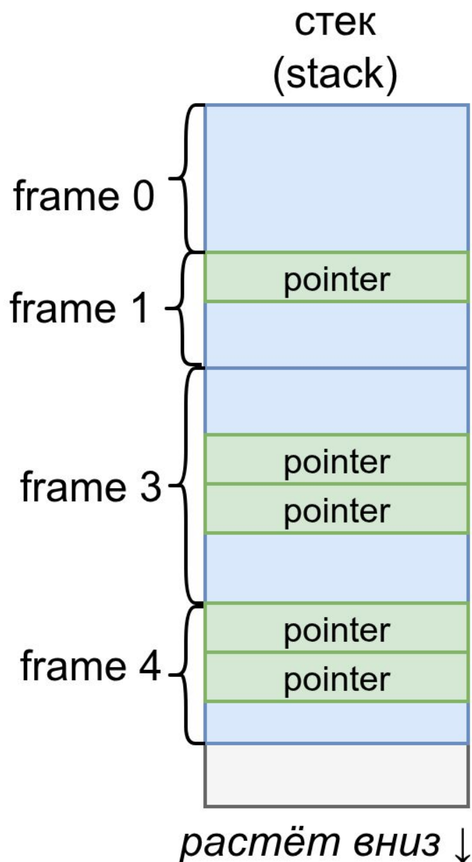
1. Указатели выровнены по границе 8 байт
2. Куча имеет фиксированный диапазон адресов
3. Обычно размер кадра функции предсказуем
   - **%esp** хранит вершину стека
   - **%ebp** может хранить начало кадра

Решение №1: консервативный GC



1. Сканируем все группы по 8 байт
   - 8 байт — размер и выравнивание указателя
2. Каждое значение считаем потенциальным указателем
   - Проверяем, попадает ли значение в диапазон кучи

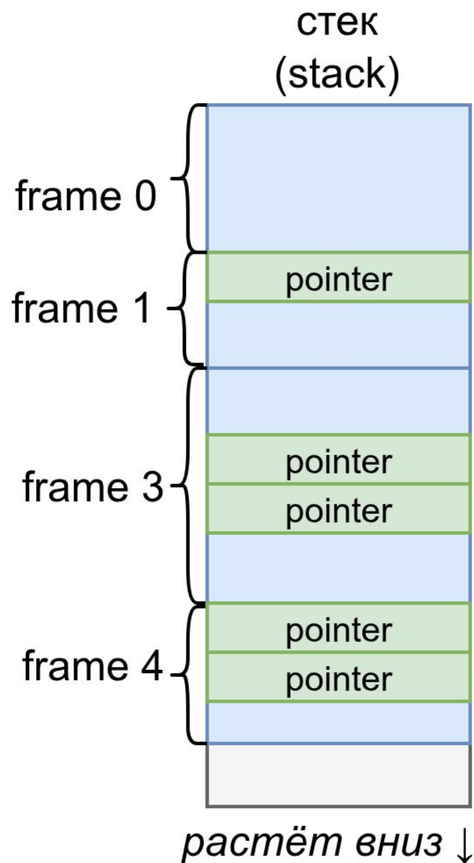
Решение №1: консервативный GC



1. Сканируем все группы по 8 байт
   - 8 байт — размер и выравнивание указателя
2. Каждое значение считаем потенциальным указателем
   - Проверяем, попадает ли значение в диапазон кучи

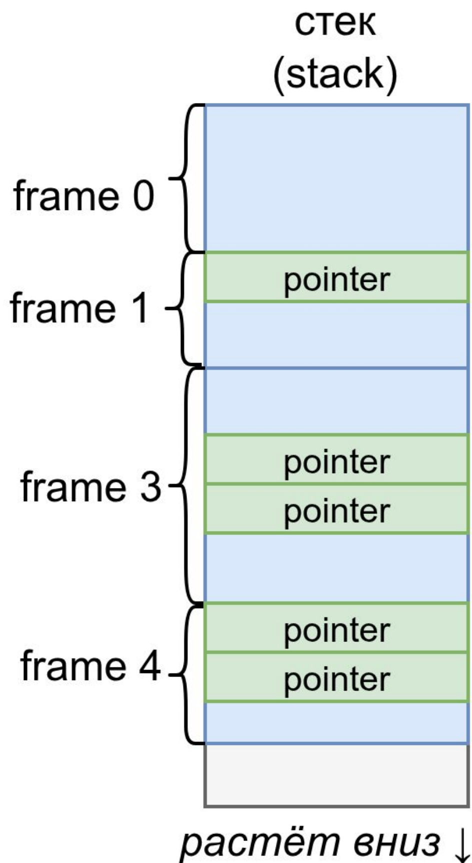
Узкое место: адресная арифметика

Решение №2: точный GC на основе stackmap



1. Компилятор создаёт stackmap для каждой функции
2. GC использует stackmap

Решение №2: точный GC на основе stackmap



1. Компилятор создаёт stackmap для каждой функции
2. GC использует stackmap
   - stackmap — это bitmap, где каждый бит показывает, является ли указателем слот в 8 байт
   - stackmap лежит в статической памяти

Конкурентная сборка мусора

Конкурентная сборка мусора

1. Короткие паузы потоков остаются
2. Любая запись указателей меняется
    - защищена барьером записи (атомарна)
    - учитывает цвет в трёхцветном алгоритме

Компилятор / интерпретатор должен поддержать конкурентный GC.

Языки программирования и сборка мусора

Языки C / C++

Работают без GC

- в C/C++ возможно:
  - ручное управление
  - подсчёт ссылок
  - дерево объектов
  - arena allocation
- в C++ есть RAII:
  - владение и move-семантика

Языки C / C++

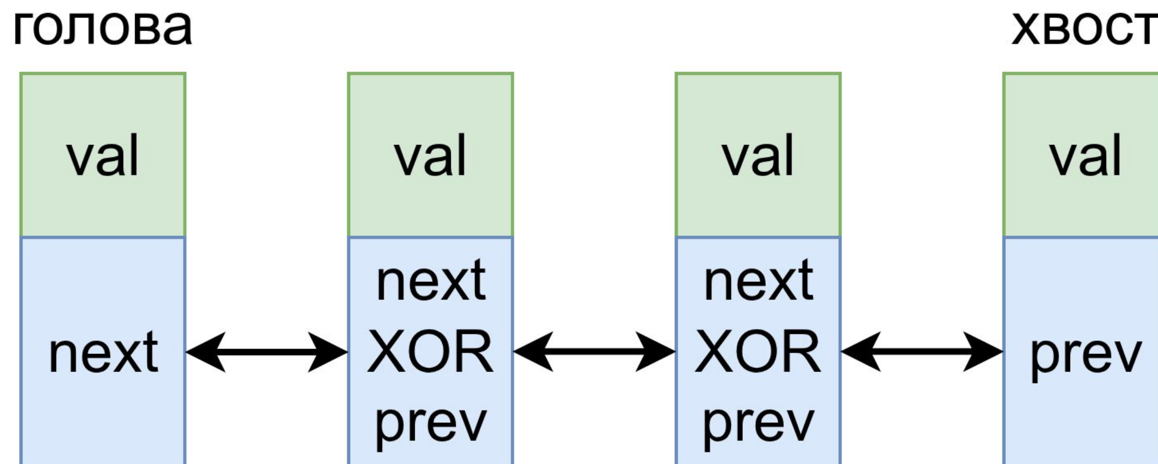
Возможен консервативный GC — например, Boehm GC

- Не терпит указателей в «закодированной» форме

Языки C / C++

Возможен консервативный GC — например, Boehm GC

- Не терпит указателей в «закодированной» форме
- Пример: XOR-связный список



Язык Go

Всегда есть GC:

1. Mark-and-sweep
    - трёхцветный алгоритм для этапа mark
2. Конкурентный
    - в Go 1.26 появился greentea — улучшен алгоритм mark
3. Неперемещающий (non-compacting)
4. Непоколенческий (non-generational)

Язык C# и среда .NET

Всегда есть GC:

1. Mark-and-sweep
2. Конкурентный — это отключаемо
3. Перемещающий (compacting)
4. Поколенческий (generational)
5. Две версии: workstation и server

Язык JavaScript и движок V8

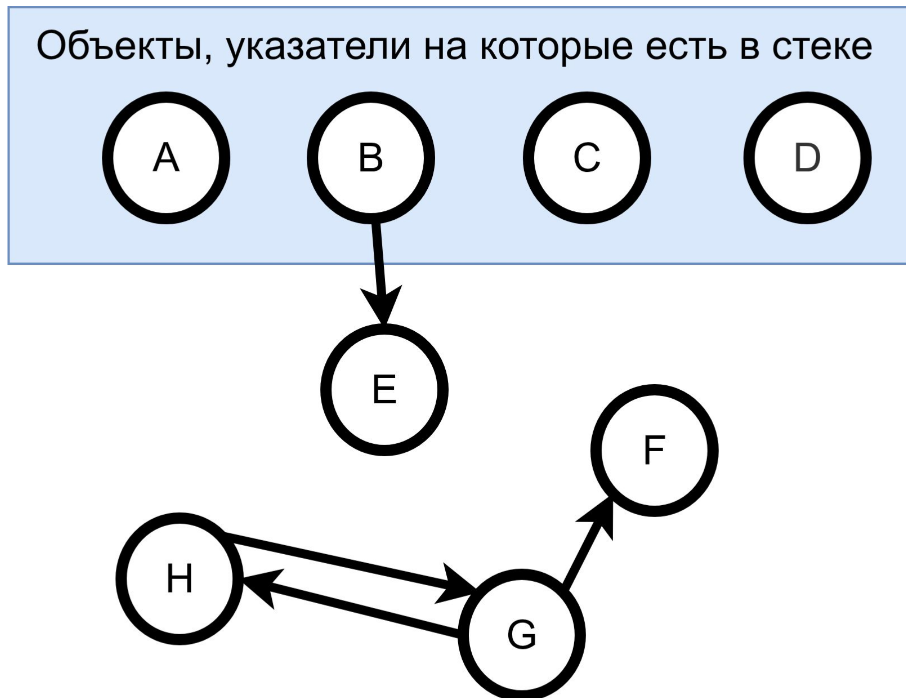
Всегда есть GC:

1. Mark-and-sweep
2. Конкурентный
3. Перемещающий (compacting)
4. Поколенческий (generational)
5. Обнаружение указателей через tagged pointer
   - Это нужно из-за динамической типизации

Подытожим

Польза сборки мусора

1. Написание кода становится проще
2. GC может найти циклы объектов



Цена сборки мусора

1. Паузы из-за сборки
   - Полный Stop the world — если GC без конкуренции
   - Частичные остановки и блокировки — если GC конкурентный
2. Перерасход памяти либо CPU
   - Частый вызов GC — пустая трата CPU
   - Редкий вызов GC — мусор занимает память
3. Меняется баланс простого и сложного
   - Обычный код писать проще
   - Высокопроизводительный — сложнее

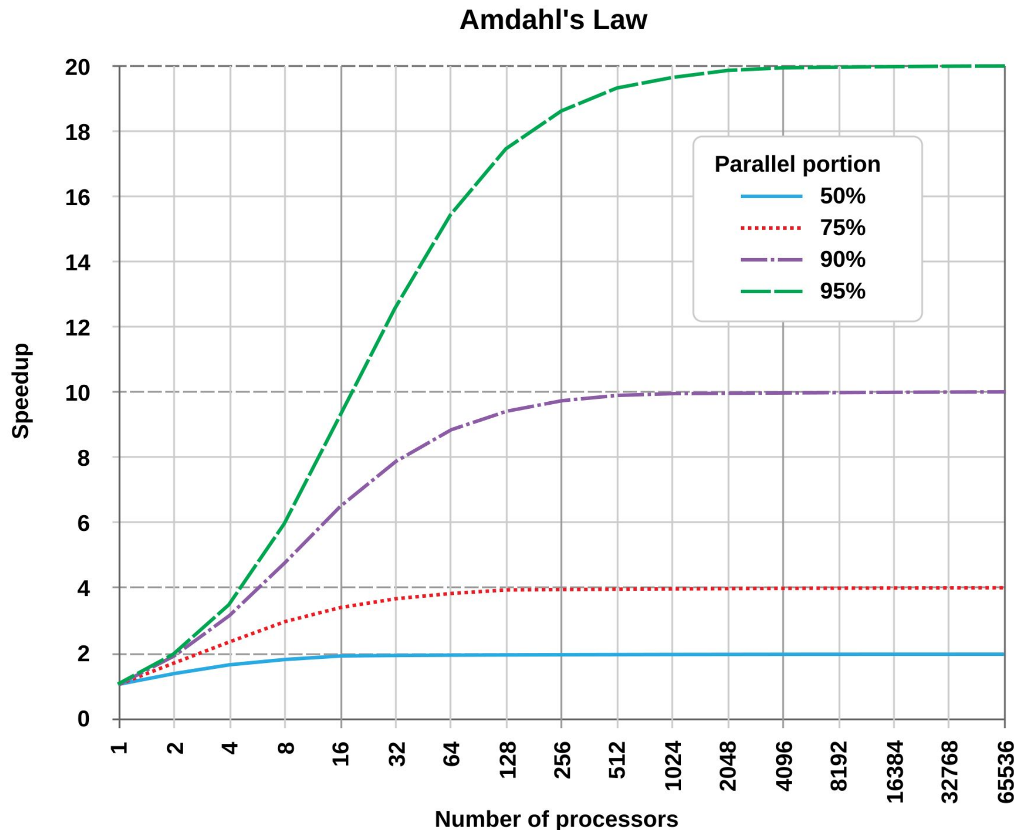
Высокопроизводительные вычисления

Выигрывают структуры данных с хорошим spatial locality

Структуры «хороших» структур:

1. Массив и его производные
   - Flat Map — сортированный массив как словарь
   - Ring Buffer — кольцевой буфер для очередей и deque
   - Куча (heap) — дерево в массиве
2. Развёрнутый связный список (Unrolled Linked List)
3. Хеш с открытой адресацией

Закон Амдала



Допустим

- **p** — число процессоров
- **a** — доля операций, выполняемых последовательно

Ускорение параллелизма равно

$$1 / (a + (1 - a) / p)$$

Пусть a=0.4, p=16

$$1 / (0.4 + (1 - 0.4) / 16)$$

$$\approx \text{в } 2.29 \text{ раз}$$

Amdahl's law - Wikipedia

Ссылки — аллокаторы

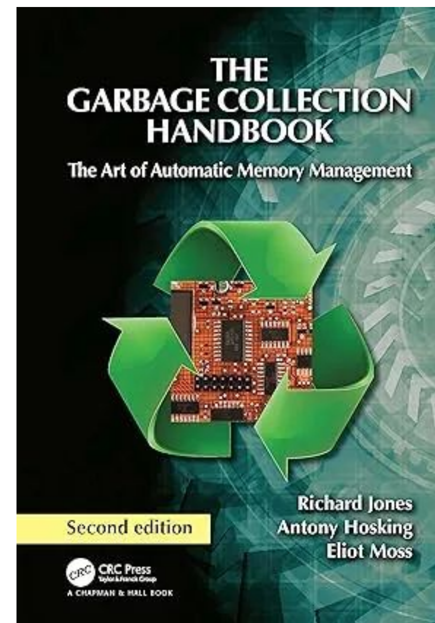
- Описание glibc malloc:
  - Understanding glibc malloc – sploitF-U-N
  - Handling native memory fragmentation of glibc
- Описание tcmalloc: TCMalloc : Thread-Caching Malloc

Ссылки — сборка мусора

- Документация LLVM по GC: Garbage Collection with LLVM
- Описание для .NET: Garbage Collection Design
- Современная версия Boehm GC: GitHub - bdwgc/bdwgc

Ссылки — сборка мусора

- Документация LLVM по GC: Garbage Collection with LLVM
- Описание для .NET: Garbage Collection Design
- Современная версия Boehm GC: GitHub - bdwgc/bdwgc
- Книга «The Garbage Collection Handbook»
    - Повлияла на Java (JVM), C# (.NET), Go



Конец!