

Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»

Выпускная квалификационная работа бакалавра

Метод автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок

Студент: Сапожков Андрей Максимович ИУ7-83Б

Научный руководитель: Строганов Юрий Владимирович

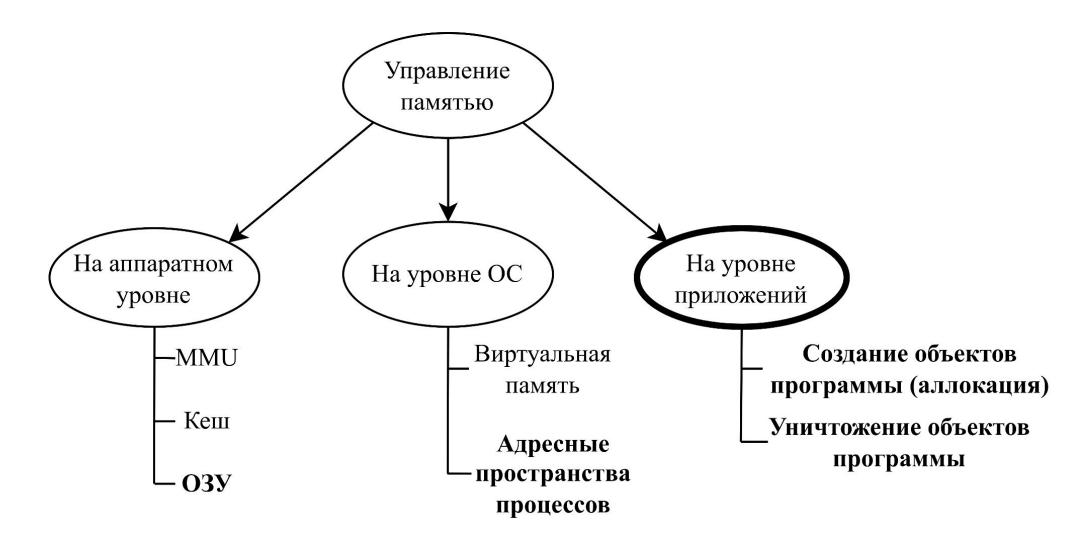
Цель и задачи

Цель — разработка метода автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок.

Задачи:

- 1. Проанализировать существующие методы распределения памяти в языках программирования с автоматической сборкой мусора.
- 2. Спроектировать метод автоматического управления памятью.
- 3. Реализовать спроектированный метод в виде подключаемой библиотеки.
- 4. Определить области применения реализованного метода.

Управление памятью



Управление памятью на уровне приложений

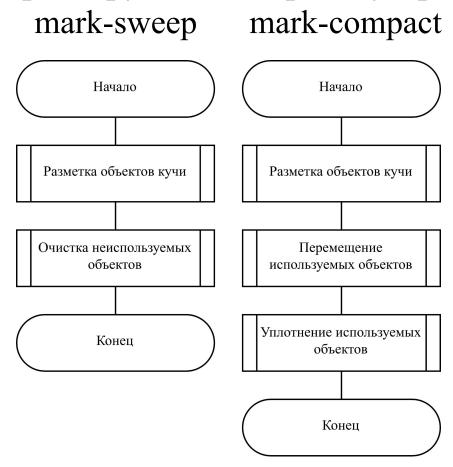
- Ручное, определяемое человеческим фактором.
- Автоматическое, определяемое алгоритмами распределения памяти.

Ручное управление памятью (С++)

Автоматическое управление памятью (Golang)

Автоматическая переработка динамически выделяемой памяти

Трассирующая сборка мусора



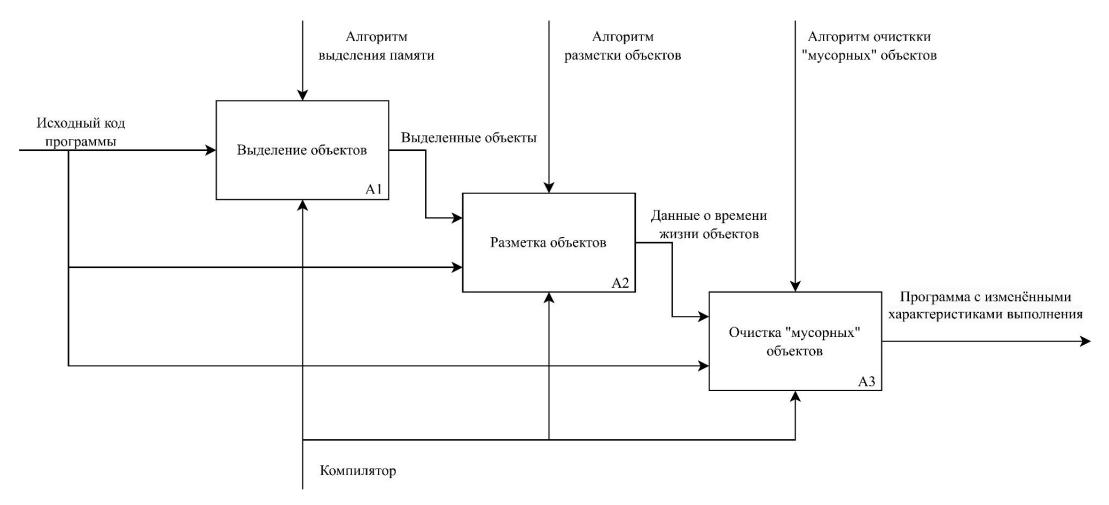
Подсчёт ссылок

- + более равномерное распределение накладных расходов;
- + устойчивость к высоким нагрузкам;
- + масштабируемость по размеру кучи;
- + не требует среды выполнения языка;
- накладные расходы на чтение/запись;
- атомарность операций;
- не выявляются циклы ссылок.

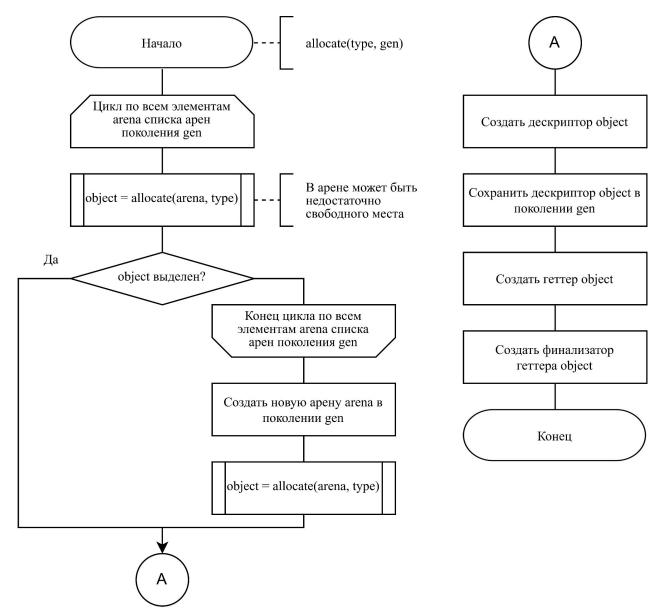
Классификация существующих решений

Язык программирования	Python	Java				JavaScript	C#	Golang
Сборщик мусора	По умолчанию	Serial	Parallel	Garbage-First	ZGC	По умолчанию	По умолчанию	По умолчанию
Разделение объектов на поколения	+	+	+	+	+	-	+	-
Отсутствие хранения вспомогательных данных в объектах	-	+	+	+	+	+	+	+
Использование конкурентной сборки мусора	+	+	+	+	+	-	+	+
Использование параллельной сборки мусора	-	-	+	+	+	-	+	+
Отсутствие остановки потоков мутатора на весь цикл сборки мусора	+	-	=	+	+	-	-	+
Количество остановок потоков мутатора за один цикл сборки мусора	1	1	1	2	1	1	1	2

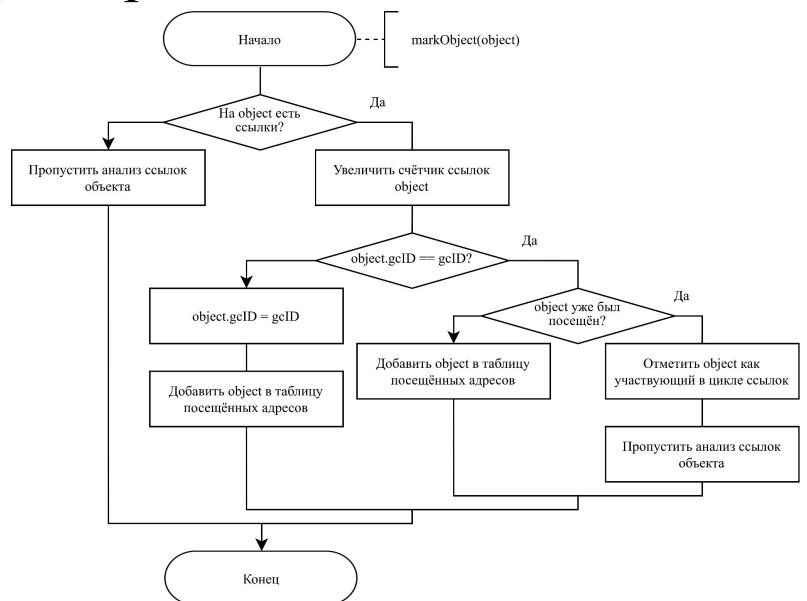
Метод автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок



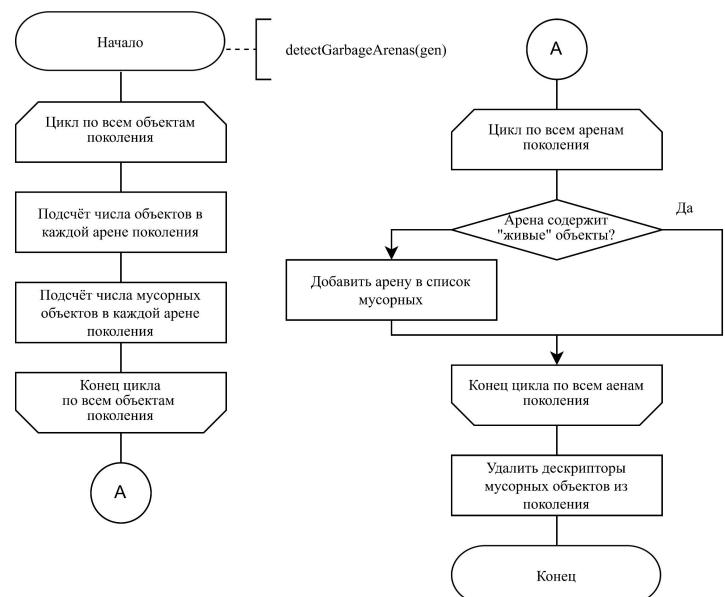
Алгоритм выделения памяти



Алгоритм разметки



Алгоритм очистки



Структура ПО

hypervisor markWorker Memory + movingGenerations: []*generation.Generation + gcID: uint64 + gcLock: sync.Mutex + gcCtx: context.Context + largeObjectGeneration: *generation.Generation + visited: map[unsafe.Pointer] bool + gcCancel: context.CancelFunc + searchMetadata: SearchFunc + mem: memory + markObject(*ObjectMetadata, bool): *ObjectMetadata, bool,bool Generation + arenaSignals: chan<- struct{} + extractMetadata(reflect.Value): *ObjectMetadata, bool + arenasMx: sync.RWMutex + gcState + analyzeObject(*ObjectMetadata): []*ObjectMetadata + arenas: []limited_arena.Arena + run() + processObject(*ObjectMetadata, bool): []*ObjectMetadata + arenaSignals: chan<- struct{} + mergeGenerations([]int, []int) + mark(<-chan *ObjectMetadata) + addresses: addressContainer[*ObjectMetadata] + calculateNewHeapTarget() + uncontrollableAddresses: addressContainer[*ObjectMetadata] + gc() ObjectMetadata gcMetadata + slices: addressContainer[*SliceMetadata] + sync.RWMutex + lastMarkID: uint64 + uncontrollableSlices: addressContainer[*SliceMetadata] qcState + Address: unsafe.Pointer + cyclicallyReferenced: bool + SearchObject(unsafe.Pointer): (*ObjectMetadata, bool) + lastCollectedGenerations: int + referenceCount: int + typeInfo: reflect.Type + SearchSliceData(unsafe.Pointer): (*SliceMetadata, bool) + curHeapSize: int + arena: *limited arena.Arena + Finalized: atomic.bool + Mark(uint64, SearchFunc) + heapTarget: int + gcMetadata + cleanUncontrollableObjects() SliceMetadata + Address(): unsafe.Pointer + detectGarbageArenas(): []*limited arena.Arena Arena + Len: int + Sweep(): (int, int) + arena: *arena.Arena + Cap: int + Size(): int + free: uintptr + ObjectMetadata + MoveTo(*Generation) + Free()

Использование метода

> go get github.com/Inspirate789/alloc

```
package main
func main() {
    object := new(int)
    println(*object)
    *object = 7
    println(*object)
Встроенные функции
```

Golang

```
println(*object.Get())
}
Реализованный метод
```

object := alloc.New[int]()

println(*object.Get())

*object.Get() = 7

import "github.com/Inspirate789/alloc"

package main

func main() {

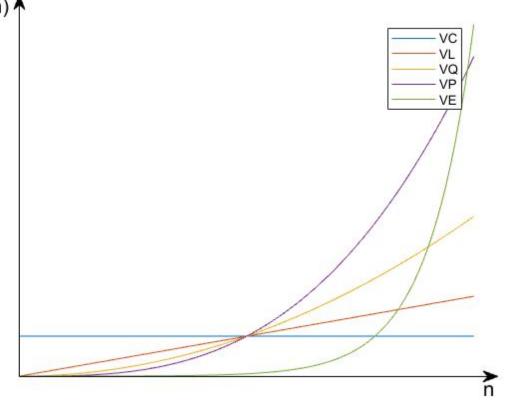
Классы алгоритмов по требованиям к дополнительной памяти

- V0: $V_t(D) = 0$ (не рассматривается) $V_t^{(n)}$
- **VC**: $V_t^{\wedge}(n) = const \neq 0$;
- VL: $V_t^{\wedge}(n) = \Theta(n)$;
- \mathbf{VQ} : $V_t^{\wedge}(n) = \Theta(n^2)$;
- **VP**: $V_t^{\wedge}(n) = \Theta(n^k)$, k > 2;
- **VE**: $V_t^{\wedge}(n) = \Theta(e^{\lambda n}), \lambda > 0$;

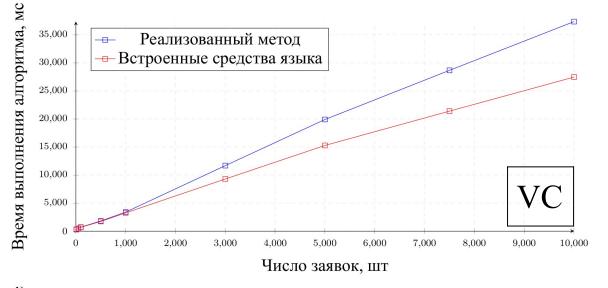
где D_n – вход алгоритма длины n,

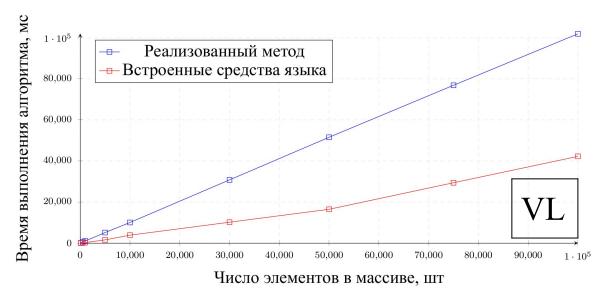
 $V_t(D)$ – дополнительная память алгоритма,

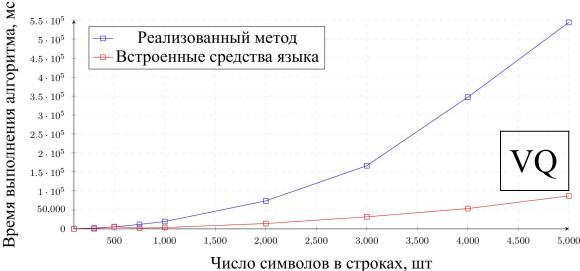
 $V_t^\wedge(n) = \max \ V_t(D) -$ дополнительная память алгоритма в худшем случае для всех входов длины n.



Проигрыш по времени выполнения алгоритмов в сравнении со встроенными средствами Golang







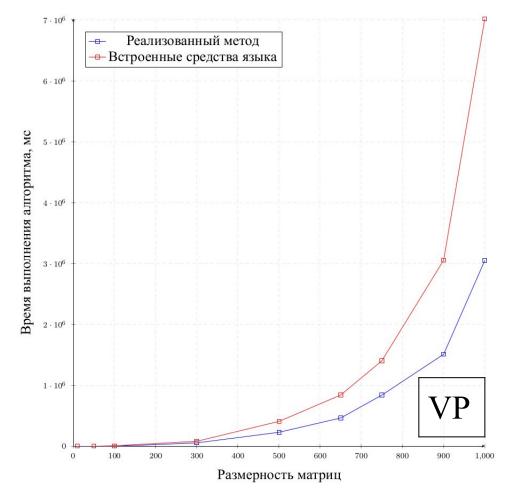
VC: фиксированная доп. память;

VL: доп. память линейно

зависит от длины входа;

VQ: доп. память квадратично зависит от длины входа.

Выигрыш по времени выполнения алгоритмов в сравнении со встроенными средствами Golang



Реализованный метод Встроенные средства языка $1.2 \cdot 10^{5}$ Время выполнения алгоритма, мс 40,000 20,000 $9 \cdot 10^{6}$ Число элементов в массиве, шт

VP: доп. память полиномиально надквадратично зависит от длины входа

VE: доп. память экспоненциально зависит от длины входа 15

Заключение

Цель работы достигнута: был разработан и реализован метод автоматического управления памятью. Все поставленные задачи были выполнены:

- Проанализированы существующие методы распределения памяти в языках программирования с автоматической сборкой мусора.
- Проведена классификация алгоритмов по требованиям к дополнительной памяти: выделены классы V0, VC, VL, VQ, VP и VE.
- Спроектирован и реализован метод автоматического управления памятью.
- Сравнительный анализ выявил области применения реализованного метода: алгоритмы класса **VP и VE**.

Дальнейшее развитие

- Исследование стабильности менеджера памяти.
- Исследование фрагментации кучи при использовании метода.
- Внедрение метода во встроенный сборщик мусора языка Golang.

Публикации

- TODO