

Интересные аллокаторы в играх

Управление аллокациями в с++

Тема заезженная, и не раз обсуждалась, все об этом знают

Ручное управления памятью new/delete (либо через RAII объекты)

- **Контроль времени жизни:**
полностью контролируем момент выделения и освобождения памяти.
В отличие от GC-языков, нет "фоновой магии".
-
-
-

Управление аллокациями в с++

Тема заезженная, и не раз обсуждалась, все об этом знают

Ручное управления памятью new/delete (либо через RAII объекты)

- **Контроль времени жизни:**
полностью контролируем момент выделения и освобождения памяти.
В отличие от GC-языков, нет "фоновой магии".
- **Детерминированная производительность:**
операции выделения/освобождения выполняются именно там, где вызваны
это критично: GC может внезапно "стопнуть мир" (stop-the-world pause) даже на десятки миллисекунд.
-
-

Управление аллокациями в c++

Тема заезженная, и не раз обсуждалась, все об этом знают

Ручное управления памятью new/delete (либо через RAII объекты)

- **Контроль времени жизни:**
полностью контролируем момент выделения и освобождения памяти.
В отличие от GC-языков, нет "фоновой магии".
- **Детерминированная производительность:**
операции выделения/освобождения выполняются именно там, где вызваны
это критично: GC может внезапно "стопнуть мир" (stop-the-world pause) даже на десятки миллисекунд.
- **Максимальный контроль над ресурсами:**
можно писать свои аллокатеры, пулы, арены
можно оптимизировать под конкретные задачи (отложенное удаление, кольцевые буферы для событий и т.п.).
-

Управление аллокациями в с++

Тема заезженная, и не раз обсуждалась, все об этом знают

Ручное управления памятью new/delete (либо через RAII объекты)

- **Контроль времени жизни:**
полностью контролируем момент выделения и освобождения памяти.
В отличие от GC-языков, нет "фоновой магии".
- **Детерминированная производительность:**
операции выделения/освобождения выполняются именно там, где вызваны
это критично: GC может внезапно "стопнуть мир" (stop-the-world pause) даже на десятки миллисекунд.
- **Максимальный контроль над ресурсами:**
можно писать свои аллокаторы, пулы, арены
можно оптимизировать под конкретные задачи (отложенное удаление, кольцевые буферы для событий и т.п.).
- **Минимальные накладные расходы:**
GC требует хранить метаданные для трекинга объектов, их связей и поколений.
память занимает ровно столько, сколько нужно структурам (плюс выравнивание и заголовок).

Управление аллокациями в с++ (минусы)

Общий подход для игр не работает

- **Ошибки управления памятью:**
утечки, double free, use-after-free, corruption
Требует дисциплины и/или умных указателей
- **Не всегда "бесплатно":**
new/delete дорогие
- **RAII + unique_ptr / shared_ptr:**
добавляют свой оверхед (счётчики, вызовы деструкторов)

Как объекты будут расположены в памяти?

Если подряд выделить несколько объектов, как они будут лежать в памяти?

Запросы на выделение памяти для объектов:

```
new Obj_A; new Obj_B; new Obj_C; new Obj_D;
```

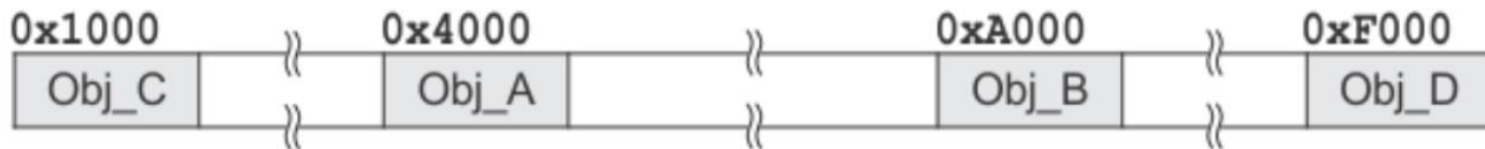
А мы не знаем как они будут в памяти расположены

Запросы на выделение памяти для объектов:

```
new Obj_A; new Obj_B; new Obj_C; new Obj_D;
```

Могут как то так лежать, а могут по другому

Фактическое размещение объектов в памяти:



Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

«Сначала ты узнаёшь ценность абстракции, потом — её стоимость, а потом — как правильно её применять»

Джозел Спольски (Joel Spolsky) - закон “утечек” абстракций

Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

- Прямое использование массива `int arr[100]` — память выделена на стеке, всё просто и дешево

Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

- Прямое использование массива `int arr[100]` — память выделена на стеке, всё просто и дешево
- `vector<int>` — динамические аллокации, хранение размера/ёмкости, возможные копирования при росте

Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

- Прямое использование массива `int arr[100]` — память выделена на стеке, всё просто и дешево
- `vector<int>` — динамические аллокации, хранение размера/ёмкости, возможные копирования при росте
- `shared_ptr<vector<int>>` — ещё дороже: помимо аллокаций вектора, идёт подсчёт ссылок на сам вектор.

Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

- Прямое использование массива `int arr[100]` — память выделена на стеке, всё просто и дешево
- `vector<int>` — динамические аллокации, хранение размера/ёмкости, возможные копирования при росте
- `shared_ptr<vector<int>>` — ещё дороже: помимо аллокаций вектора, идёт подсчёт ссылок на сам вектор.
- это все ещё тот же самый указатель на массив из 100 чисел, только с кучей “обвеса”

Цена абстракций

Абстрагирование от управления памятью обходится тем дороже, чем больше абстракций мы туда добавили

- Прямое использование массива `int arr[100]` — память выделена на стеке, всё просто и дешево
- `vector<int>` — динамические аллокации, хранение размера/ёмкости, возможные копирования при росте
- `shared_ptr<vector<int>>` — ещё дороже: помимо аллокаций вектора, идёт подсчёт ссылок на сам вектор.
- это все ещё тот же самый указатель на массив из 100 чисел, только с кучей “обвеса”

Аллокатор общего назначения в движении

Стандартный аллокатор лишает нас всех преимуществ ручного управления памятью

- **Потеря предсказуемости времени работы**
Общие аллокаторы оптимизированы под широкий спектр сценариев
-
-
-

Аллокатор общего назначения в движении

Стандартный аллокатор лишает нас всех преимуществ ручного управления памятью

- **Потеря предсказуемости времени работы**

Общие аллокаторы оптимизированы под широкий спектр сценариев

- **Фрагментация памяти**

Универсальный аллокатор гибкий - это всегда ведёт к «дыркам» в куче при частых аллокациях и деаллокациях разного размера

-

-

Аллокатор общего назначения в движке

Стандартный аллокатор лишает нас всех преимуществ ручного управления памятью

- **Потеря предсказуемости времени работы**

Общие аллокаторы оптимизированы под широкий спектр сценариев

- **Фрагментация памяти**

Универсальный аллокатор гибкий - это всегда ведёт к «дыркам» в куче при частых аллокациях и деаллокациях разного размера

- **Избыточные накладные расходы**

Стандартный аллокатор хранит много метаданных для каждого блока (размер, ссылки и т. д.).

-

Аллокатор общего назначения в движке

Стандартный аллокатор лишает нас всех преимуществ ручного управления памятью

- **Потеря предсказуемости времени работы**

Общие аллокаторы оптимизированы под широкий спектр сценариев

- **Фрагментация памяти**

Универсальный аллокатор гибкий - это всегда ведёт к «дыркам» в куче при частых аллокациях и деаллокациях разного размера

- **Избыточные накладные расходы**

Стандартный аллокатор хранит много метаданных для каждого блока (размер, ссылки и т. д.).

- **Отсутствие контроля над расположением данных**

Общий аллокатор не знает как работать с “нашими” данными

Кто-то «живет один кадр» или «ресурсы уровня загружаются и освобождаются одним блоком».

Аллокатор общего назначения в движении

Стандартный аллокатор лишает нас всех преимуществ ручного управления памятью

- **Потеря предсказуемости времени работы**
Общие аллокаторы оптимизированы под широкий спектр сценариев
- **Фрагментация памяти**
Универсальный аллокатор гибкий - это всегда ведёт к «дыркам» в куче при частых аллокациях и деаллокациях разного размера
- **Избыточные накладные расходы**
Стандартный аллокатор хранит много метаданных для каждого блока (размер, ссылки и т. д.).
- **Отсутствие контроля над расположением данных**
Общий аллокатор не знает как работать с «нашими» данными
Кто-то «живет один кадр» или «ресурсы уровня загружаются и освобождаются одним блоком».
- Платим лишнюю цену за гибкость, которая в игре не нужна.

Какой колобок испортил память?

const **char*** name1 = "kolobok";

std::string name2 = "kolobok";

eastl::string name3 = "kolobok";

frozen::string<16> name4 = "kolobok";

const **char[]** name5 = "kolobok";

Какой колобок испортил память?

```
const char* name1 = "kolobok"; // нет, данные в .data  
  
std::string name2 = "kolobok";  
  
eastl::string name3 = "kolobok";  
  
frozen::string<16> name4 = "kolobok";  
  
const char[] name5 = "kolobok";
```

Какой колобок испортил память?

```
const    char*    name1    =    "kolobok";        //    нет,    данные    в    .data

std::string    name2    =    "kolobok";        //    чаще    да,    если    нет    SS0

eastl::string                name3                =                "kolobok";

frozen::string<16>                name4                =                "kolobok";

const                char[]                name5                =                "kolobok";
```

Какой колобок испортил память?

```
const char* name1 = "kolobok"; // нет, данные в .data

std::string name2 = "kolobok"; // чаще да, если нет SS0

eastl::string name3 = "kolobok"; // есть SS0, нет аллокации

frozen::string<16> name4 = "kolobok";

const char[] name5 = "kolobok";
```

Какой колобок испортил память?

```
const char* name1 = "kolobok"; // нет, данные в .data  
  
std::string name2 = "kolobok"; // чаще да, если нет SSO  
  
eastl::string name3 = "kolobok"; // есть SSO, нет аллокации  
  
frozen::string<16> name4 = "kolobok"; // статическая строка, нет  
  
const char[] name5 = "kolobok";
```


Какой колобок испортил память?

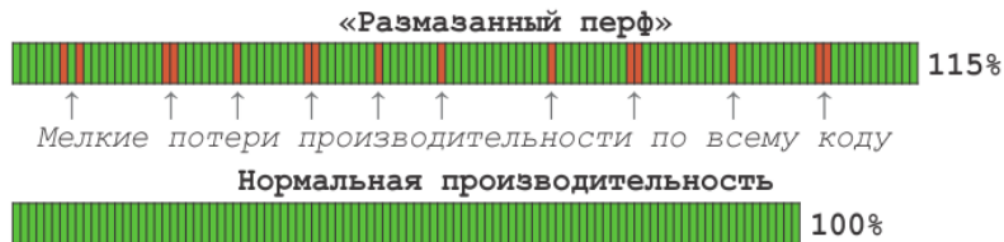
```
const char* name1 = "kolobok"; // нет, данные в .data  
  
std::string name2 = "kolobok"; // чаще да, если нет SSO  
  
eastl::string name3 = "kolobok"; // есть SSO, нет аллокации  
  
frozen::string<16> name4 = "kolobok"; // статическая строка, нет  
  
const char[] name5 = "kolobok"; // нет, данные на стеке
```

Не спешите писать свой аллокатор

Я не призываю вас встать на путь ручного управления памятью, ибо он будет усеян ловушками, граблями и утечками, но разработчик в итоге оказывается перед выбором: либо довериться системному аллокатору и столкнуться с проблемами “размазанного перфа”, когда код написан правильно, но работает не быстро, либо взять всё в свои руки, создавая собственные механизмы выделения и освобождения ресурсов.

Не спешите писать свой аллокатор

Я не призываю вас встать на путь ручного управления памятью, ибо он будет усеян ловушками, граблями и утечками, но разработчик в итоге оказывается перед выбором: либо довериться системному аллокатору и столкнуться с проблемами “размазанного перфа”, когда код написан правильно, но работает не быстро, либо взять всё в свои руки, создавая собственные механизмы выделения и освобождения ресурсов.



Нам нужен свой аллокатор, если . . .

- **Нужен контроль времени выделения и освобождения памяти**
нельзя допускать случайных пауз из-за фрагментации или долгого освобождения памяти стандартным `malloc/new`.
-
-
-
-

Нам нужен свой аллокатор, если . . .

- **Нужен контроль времени выделения и освобождения памяти**
нельзя допускать случайных пауз из-за фрагментации или долгого освобождения памяти стандартным `malloc/new`.
- **Хотим избежать фрагментации памяти**
Часто создаем и удаляем мелкие объекты (частицы, пули, эффекты)
-
-
-

Нам нужен свой аллокатор, если . . .

- **Нужен контроль времени выделения и освобождения памяти**
нельзя допускать случайных пауз из-за фрагментации или долгого освобождения памяти стандартным `malloc/new`.
- **Хотим избежать фрагментации памяти**
Часто создаем и удаляем мелкие объекты (частицы, пули, эффекты)
- **Хотим видеть предсказуемое использование памяти**
Для консолей и мобильных есть строгие лимиты
Свой аллокатор позволяет заранее выделить большой блок (например, арену на 64 МБ) и раздать память внутри неё
-
-

Нам нужен свой аллокатор, если . . .

- **Нужен контроль времени выделения и освобождения памяти**
нельзя допускать случайных пауз из-за фрагментации или долгого освобождения памяти стандартным `malloc/new`.
- **Хотим избежать фрагментации памяти**
Часто создаем и удаляем мелкие объекты (частицы, пули, эффекты)
- **Хотим видеть предсказуемое использование памяти**
Для консолей и мобильных есть строгие лимиты.
Свой аллокатор позволяет заранее выделить большой блок (например, арену на 64 МБ) и раздать память внутри неё
- **Знаем как работают наши данные**
Арена для объектов, живущих один кадр (частица),
линейный аллокатор для уровня (всё выделяется при загрузке, освобождается сразу после выхода)
-

Нам нужен свой аллокатор, если . . .

- **Нужен контроль времени выделения и освобождения памяти**
нельзя допускать случайных пауз из-за фрагментации или долгого освобождения памяти стандартным `malloc/new`.
- **Хотим избежать фрагментации памяти**
Часто создаем и удаляем мелкие объекты (частицы, пули, эффекты)
- **Хотим видеть предсказуемое использование памяти**
Для консолей и мобильных есть строгие лимиты.
Свой аллокатор позволяет заранее выделить большой блок (например, арену на 64 МБ) и раздать память внутри неё
- **Знаем как работают наши данные**
Арена для объектов, живущих один кадр (частица),
линейный аллокатор для уровня (всё выделяется при загрузке, освобождается сразу после выхода)
- **Хотим работать очень быстро**
Собственные аллокаторы позволяют контролировать, **как именно данные лежат в памяти**
рядом ли сущности одного типа и не пересекаются ли они с «шумными» структурами



junior using custom allocators



senior using custom allocator

Аллокаторы бывают разные

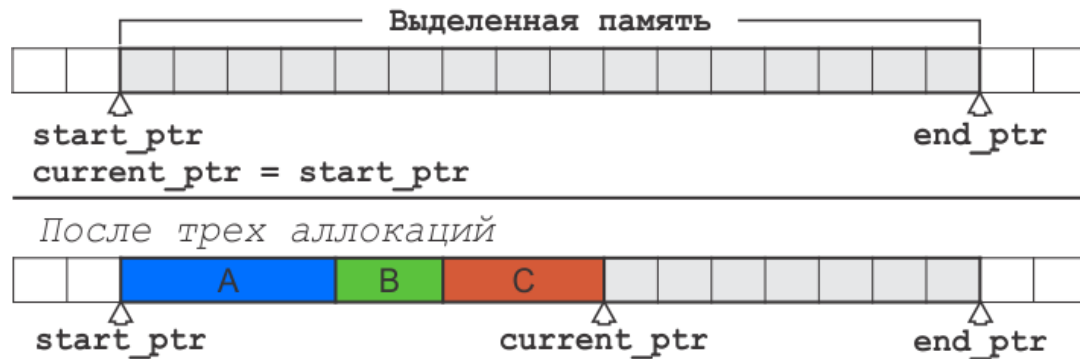
Назовите известные вам аллокаторы...

Линейный аллокатор (золото игростроя)

Самый простой и эффективный механизм управления памятью. Используется там, где требуется высокая производительность и минимальные накладные расходы

Линейный аллокатор (золото игростроя)

Самый простой и эффективный механизм управления памятью. Используется там, где требуется высокая производительность и минимальные накладные расходы



Линейный аллокатор

```
struct LinearAllocator {  
    void* base_pointer;  
    size_t size, offset;  
  
    void* allocate(size_t alloc_size) {  
        if (offset + alloc_size > size) return nullptr;  
        void* ptr = static_cast<char*>(base_pointer) + offset;  
        offset += alloc_size;  
        return ptr;  
    }  
  
    void reset() { offset = 0; }  
};
```

Линейный аллокатор (где применяется)

- для временных объектов на стеке

```
char stack_buffer[1024];  
// Линейный аллокатор поверх буфера  
std::pmr::monotonic_buffer_resource stack_resource{ stack_buffer, sizeof(stack_buffer) };  
// pmr-вектор, использующий наш аллокатор  
std::pmr::vector<int> v{ &stack_resource };  
for (int i = 0; i < 100; ++i) {  
    v.push_back(i);  
} // нет аллокаций
```

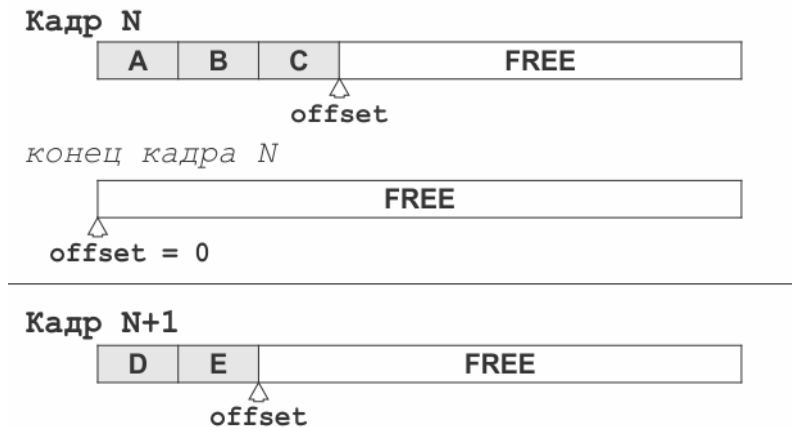
- для фрейма
очень быстрый, delete не нужен, пользоваться надо осторожно
- для уровня
статика уровня, разрушаемые объекты, выгружаем сразу блоками

Фреймовый аллокатор

Специализированный вид linear-аллокатора, который используется для управления памятью в циклических или временных алгоритмах, где данные часто выделяются и освобождаются в рамках определённого времени

Фреймовый аллокатор

Специализированный вид linear-аллокатора, который используется для управления памятью в циклических или временных алгоритмах, где данные часто выделяются и освобождаются в рамках определённого времени



Фреймовый аллокатор (где используется)

- Локальный аллокатор для потока или задачи
Нет накладных расходов, но очень небольшая область применения
Надо закладывать $\times 1.5$ реально используемой памяти
- Временные буферы
Команды рендеринга, данные для шейдеров, отладочная геометрия
- Очереди событий
Действия пользователя, ивенты клавиатуры, контроллеров
- AI
Временные данные для поиска пути, поведения NPC, реакций и т.д., которые устареют к следующему кадру

Двойной фреймовый аллокатор

Развитие идеи фрейм-аллокатора к управлению временной памятью, который сочетает в себе простоту и высокую производительность. Основная идея заключается в том, чтобы использовать две независимые области памяти (региона), между которыми происходит чередование от кадра к кадру

Двойной фреймовый аллокатор

Развитие идеи фрейм-аллокатора к управлению временной памятью, который сочетает в себе простоту и высокую производительность. Основная идея заключается в том, чтобы использовать две независимые области памяти (региона), между которыми происходит чередование от кадра к кадру

Кадр N

Пул 1 (активный)

A	B	C	FREE
---	---	---	------

Пул 2 (неактивный)

D	E	FREE
---	---	------

 ← данные с кадра N-1

Кадр N+1

Пул 1 (неактивный)

A	B	C	FREE
---	---	---	------

 ← данные, сохраненные для кадра N+1

Пул 2 (активный)

F	G	FREE
---	---	------

Кадр N+2

Пул 1 (активный)

H	FREE
---	------

 ← пул сброшен, данные кадра N удалены

Пул 2 (неактивный)

F	G	FREE
---	---	------

 ← данные, сохраненные для кадра N+2

Двойной фреймовый аллокатор (пример)

```
class DoubleFrameAllocator {
    FrameAllocator allocators[2];
    uint32 frame = 0;

public:
    void* allocate(size_t size) {
        return allocators[frame % 2].allocate(size);
    }

    void nextFrame() {
        allocators[currentAllocator].reset();
        ++frame;
    }
};
```

Двойной фреймовый аллокатор (где используется)

- **GPU команды**
пока GPU обрабатывает команды кадра N, CPU готовит команды для кадра N+1
- **Uniform буферы**
данные шейдеров должны оставаться валидными пока GPU их не обработает
- **Vertex/Index буферы**
данные предыдущего кадра еще нужны для стриминга
- **Job system**
задачи предыдущего кадра могут завершаться во время следующего
- **AI**
для работы используются данные с предыдущего кадра (committed data), чтобы избежать неопределенности в поведении и иметь фиксированное состояние сцены для всех акторов

Стековый аллокатор (золото игростроя)

- ⊙ Простой и эффективный подход
- ⊙ Скорость, предсказуемость
и минимальные накладные расходы.
- ⊙ Память выделяется и освобождается
в строгом порядке LIFO
- ⊙ Последний выделенный блок памяти
должен быть освобождён первым.

Стековый аллокатор (золото игростроя)

- ⦿ Простой и эффективный подход
- ⦿ Скорость,
и минимальные накладные расходы.
- ⦿ Память выделяется
в строгом порядке LIFO
- ⦿ Последний выделенный
должен быть освобождён первым.



Стековый аллокатор (применение)

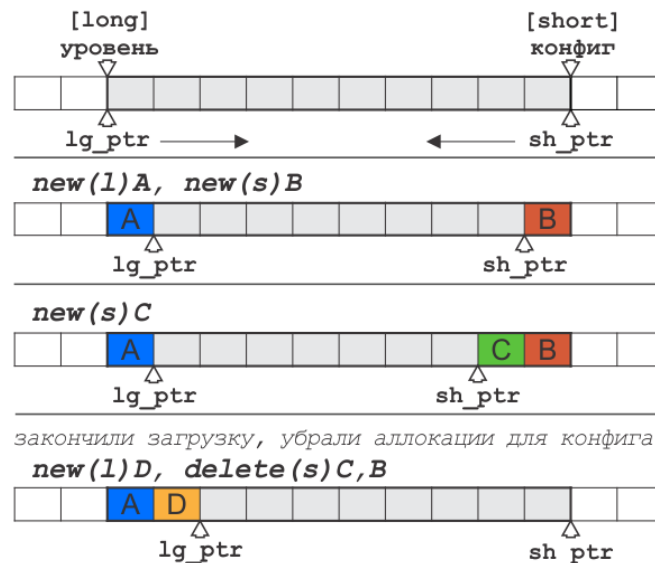
- **Рендеринг**
вложенные render passes, push/pop состояний OpenGL/Vulkan
компиляция шейдеров, хранение вложенных include структур и хедеров
- **Обход дерева сцены**
depth-first обход с сохранением состояния
- **Парсинг и загрузка ресурсов**
XML/JSON парсеры - вложенные элементы и объекты
- **Pathfinding**
рекурсивные алгоритмы типа A*, JPS (Jump Point Search), BucketNav (D*)

Двойной стековый аллокатор

Полезное расширение классического стекового аллокатора, в задачах с загрузкой больших объемов смешанных данных

Двойной стековый аллокатор

Полезное расширение классического стекового аллокатора, в задачах с загрузкой больших объемов смешанных данных

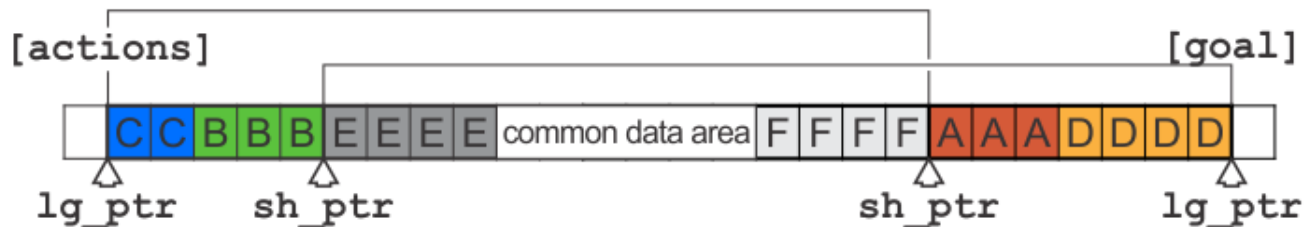


Тройной стековый аллокатор

Пик развития стековых аллокаторов - механизм тройного стека. Широкое применение получил в архитектуре AI NPC серии The Sims. Механизм двойной стековой аллокации был доработан для эффективного управления памятью при обработке сложных поведенческих паттернов симуляции персонажей

Тройной стековый аллокатор

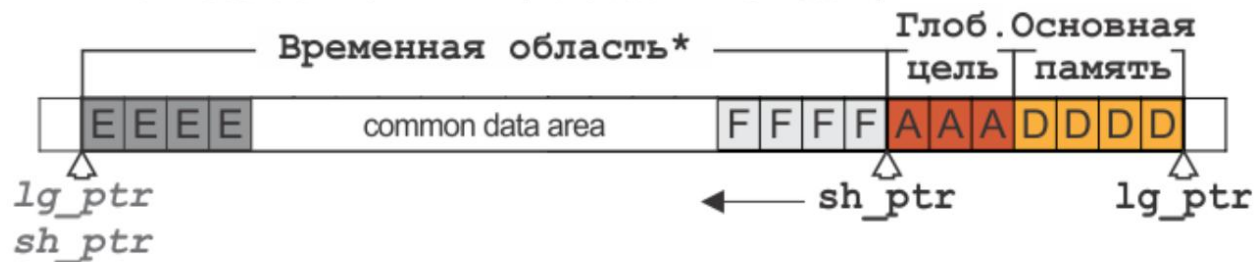
Пик развития стековых аллокаторов - механизм тройного стека. Широкое применение получил в архитектуре AI NPC серии The Sims. Механизм двойной стековой аллокации был доработан для эффективного управления памятью при обработке сложных поведенческих паттернов симуляции персонажей



Механизм тройного стека (шаг первый)

1. GOING_TO_WORK

Активная область: $A \Rightarrow D$ (глобальные цели)



*Временная область:

- Текущий путь: дом \rightarrow офис
- Промежуточные вычисления навигации
- Проверка препятствий на маршруте

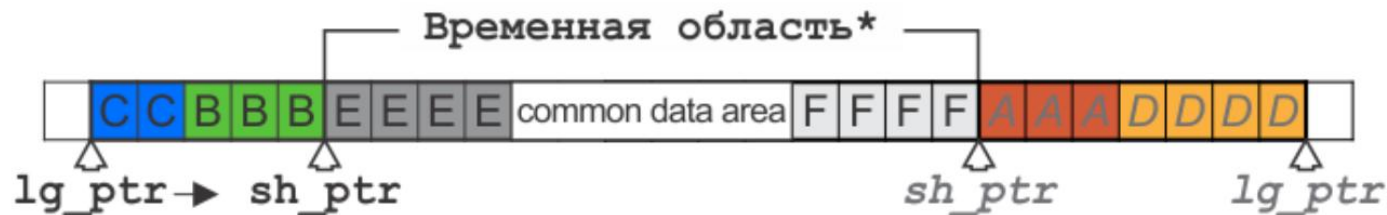
ОБЛАСТЬ A: [Цель: "Работа"] \leftarrow растет влево при необходимости

ОБЛАСТЬ D: [План дня, История действий] \leftarrow основная память

Механизм тройного стека (шаг второй)

2. URGENT_NEED_BATHROOM

Переключение на область C=>A для обработки короткой цели



*Временная область:

- Поиск ближайшей ванной
- Прерывание текущего маршрута
- Расчет времени на биологическую потребность

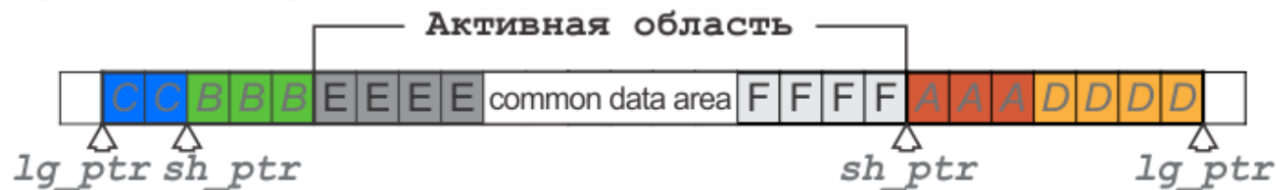
ОБЛАСТЬ С: [Цель: "Ванна", Приоритет: ВЫСОКИЙ] ← растет вправо

ОБЛАСТЬ В: [Сохранение состояния "идти на работу"]

Механизм тройного стека (шаг третий)

3. USING_BATHROOM

Продолжение работы в области E=>F



ОБЛАСТЬ E=>F:

[Локальная цель: "Туалет" - ВЫПОЛНЯЕТСЯ]:

- Анимация использования туалета
- Таймер выполнения действия
- Обновление потребностей сима (комфорт +80)
- Проверка завершения действия

Заморожены до завершения локальной цели:

ОБЛАСТИ C=>B, A=>D: *[Состояние глоб. цели для восстановления]*

Тройной стековый аллокатор (применения)

- В играх теперь встречается редко, памяти хватает
- **Встроенные системы и IoT**
Микроконтроллеры - эффективное использование ограниченной памяти с двукратной экономией
Automotive ECU – обработка CAN, диагностика и конфигурации
- **Промышленные контроллеры**
PLC системы - обработка входных сигналов, логики управления и HMI интерфейса
SCADA системы - буферизация данных и архивирование в реальном времени
- **Робототехника**
сенсорная обработка, планирование движений, AI

Пул памяти

Проанализировав недостатки линейных аллокаторов и аллокаторов общего назначения, можно попробовать их исправить для объектов одинакового размера с похожим жизненным циклом

Можно выделять и управлять блоками фиксированного размера

Пул памяти

Проанализировав недостатки линейных аллокаторов и аллокаторов общего назначения, можно их исправить для объектов одинакового размера с похожим жизненным циклом

Будем выделять блоки фиксированного размера

Непрерывный блок памяти (512 байт) <i>(разделен на блоки фиксированного размера)</i>							
Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
FREE	FREE	FREE	FREE	FREE	FREE	FREE	FREE

List: 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → NULL

Пул памяти

Проанализировав недостатки линейных аллокаторов и аллокаторов общего назначения, можно их исправить для объектов одинакового размера с похожим жизненным циклом

Будем выделять блоки фиксированного размера

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
NPC_A	NPC_B	NPC_C	FREE	FREE	FREE	FREE	FREE

Free List: 3 → 4 → 5 → 6 → 7 → NULL

Used List: 0 → 1 → 2

NPC_A: { id: 001, pos: (10,20), health: 100, ai_state: PATROL }

NPC_B: { id: 002, pos: (15,25), health: 100, ai_state: IDLE }

NPC_C: { id: 003, pos: (8,18), health: 100, ai_state: GUARD }

Пул памяти

Преимущество пула в возможности переиспользования объектов. Вместо полного уничтожения и пересоздания объектов можно создать поведение, когда объекты не удаляются физически, а переводятся в неактивное состояние и возвращаются в пул для последующего использования.

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
NPC_A	NPC_B	NPC_C	FREE	FREE	FREE	FREE	FREE

Free List: 3 → 4 → 5 → 6 → 7 → NULL

Used List: 0 → 1 → 2

NPC_A: { id: 001, pos: (10,20), health: 100, ai_state: PATROL }

NPC_B: { id: 002, pos: (15,25), health: 100, ai_state: IDLE }

NPC_C: { id: 003, pos: (8,18), health: 100, ai_state: GUARD }

Пул памяти

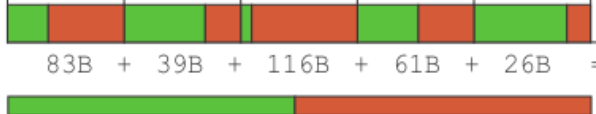
Важно знать свои структуры данных, чтобы пул не приводил к перерасходу памяти

Пул для объектов по 128 байт:

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
128 B	128 B	128 B	128 B	128 B	128 B	128 B	128 B

Реальное использование:

Obj_A	Obj_B	Obj_C	Obj_D	Obj_E	FREE	FREE	FREE
45 B	89 B	12 B	67 B	102 B			



83B + 39B + 116B + 61B + 26B = 325B ← потеряно от внутренней фрагментации

✗ Потери памяти: 325 байт из 640 = 50.8%

Пул памяти (применение)

- **Particle emitters**

частицы одинакового размера создаются и уничтожаются массово
искры, дым, огонь, магические эффекты

- **Weather systems**

капли дождя, снежинки, листья

- **Destruction effects**

обломки, декали (следы от пуль), осколки при разрушении

- **Bullets и projectiles**

снаряды, стрелы, заклинания

- **Pickups**

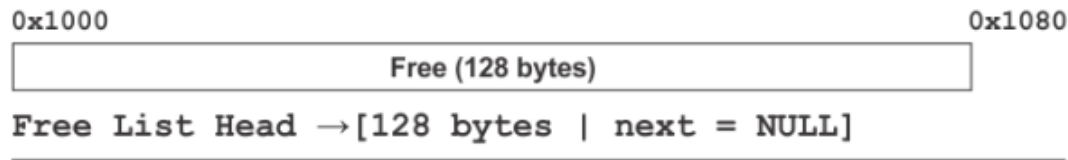
монеты, бонусы, предметы для сбора

Аллокатор со списком свободных блоков

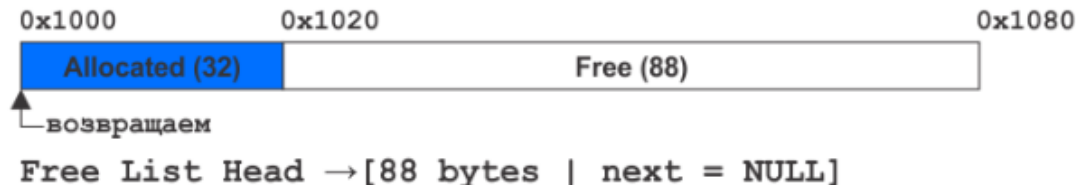
Пулы памяти позволяют выделять и освобождать память в произвольном порядке за постоянное время $O(1)$, с минимальной внешней фрагментацией — все блоки имеют фиксированный размер и следующий объект всегда поместится в любое свободное место. Но внутренняя фрагментация остается проблемой, когда объект меньше размера блока - блоки могут быть только одинакового размера. А что если нужны объекты разных размеров?

Аллокатор со списком свободных блоков

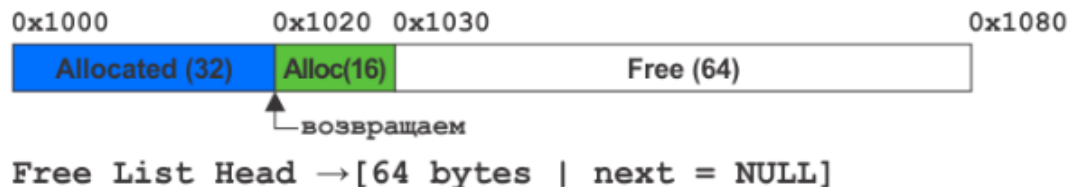
1. Начальное состояние (вся память свободна)



2. Выделение 32 байт (`malloc(32)`)



3. Выделение 16 байт (`malloc(16)`)



Аллокатор со списком свободных блоков

4. Выделение 48 байт (`malloc(48)`)



Free List Head → [8 bytes | next = NULL]

5. Освобождение среднего блока (`free(0x1020)`)



Free List Head → [8 bytes | next → 16 bytes]

6. Освобождение первого блока (`free(0x1000)`) → Объединение



Free List Head → [48 bytes | next → 8 bytes]

Аллокатор со списком свободных блоков (применение)

В чистом виде применяется достаточно редко, иногда используются для:

- **Texture memory**
управление видеопамятью для текстур разных разрешений
- **String interning**
строки разной длины в играх
- **Heightmaps**
карты высот разного разрешения
- **Локальные решения**
для оптимизации производительности

Аллокатор со списком блоков под разные размеры

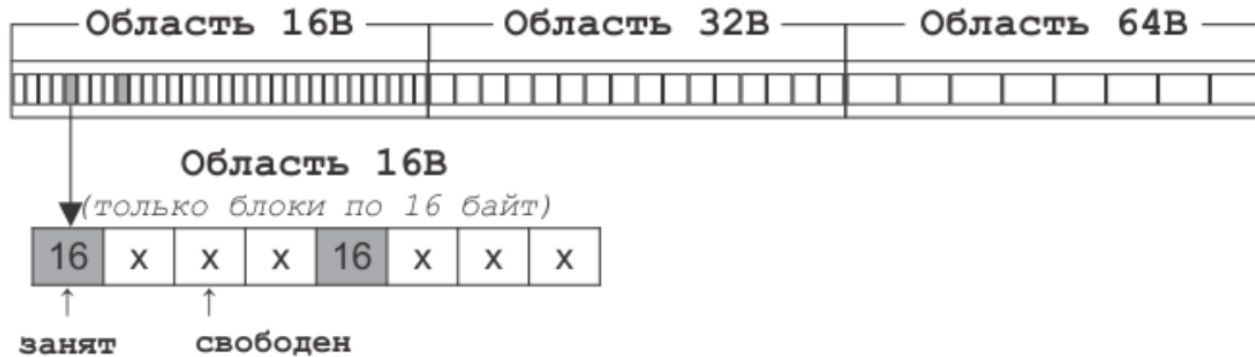
Развитием идей таких аллокаторов стало разделение объектов по группам
размеров

Вся свободная память разделяется на несколько подмножеств на основе
размеров блоков

Каждое подмножество содержит блоки определенного размера:
8-16 байт, 17-32 байта, 33-64 байта и так далее

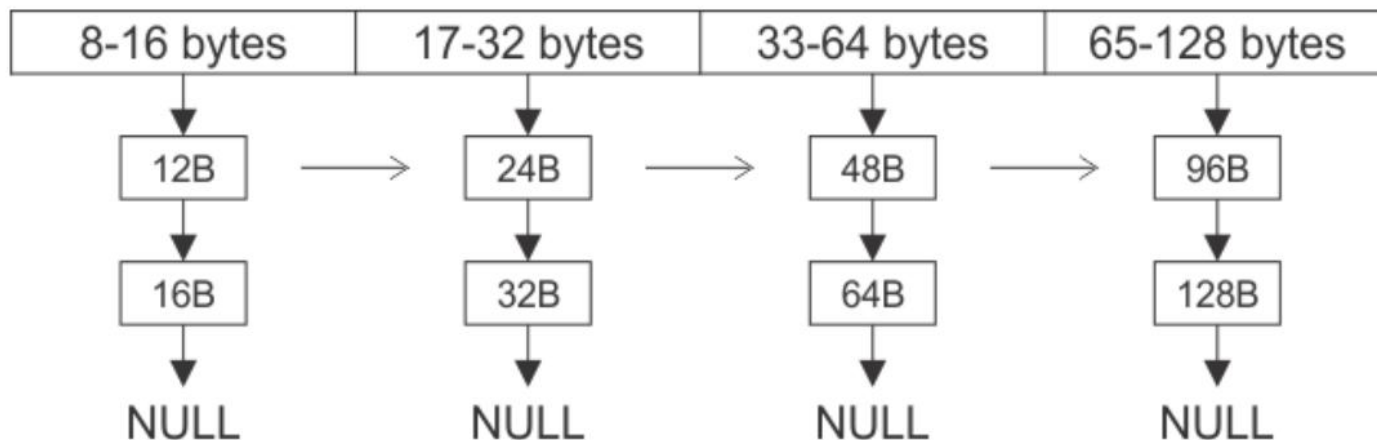
Аллокатор со списком блоков под разные размеры

Вся свободная память разделяется на несколько подмножеств на основе
размеров блоков



Аллокатор со списком блоков под разные размеры

Каждое подмножество содержит блоки определенного размера:
8-16 байт, 17-32 байта, 33-64 байта и так далее



Аллокатор со списком блоков под разные размеры (применение)

- **General**
основной аллокатор движка с размерными классами (8, 16, 32, 64, 128 байт и т.д.)
purpose **heap**
- **Small**
мелкие объекты, математические структуры (Vector3, Matrix4x4, Quaternion)
структуры с предсказуемыми размерами
object **allocation**
- **String**
строки разбитые по размерным классам (16, 32, 64, 128, 256 символов)
management
- **Физика**
примитивы коллизий (Sphere, Box, Capsule, ConvexHull)

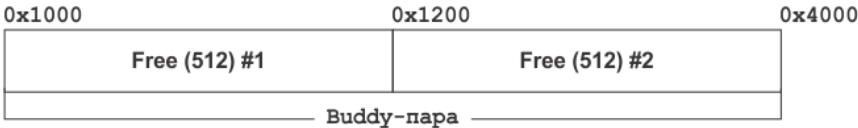
Buddy allocator

Аллокатор основан на принципе фрактализации доступного объема памяти: все выделяемые блоки имеют размер, равный степени двойки (4, 8, 16, 32, 64, 128 байт и так далее)

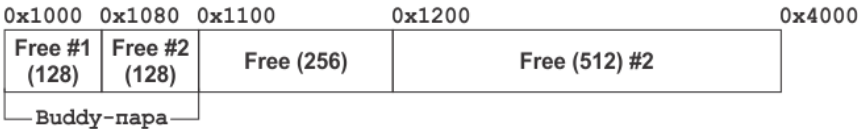
1. Начальное состояние (память 1024 байта = 2^{10})



2. Первое разделение



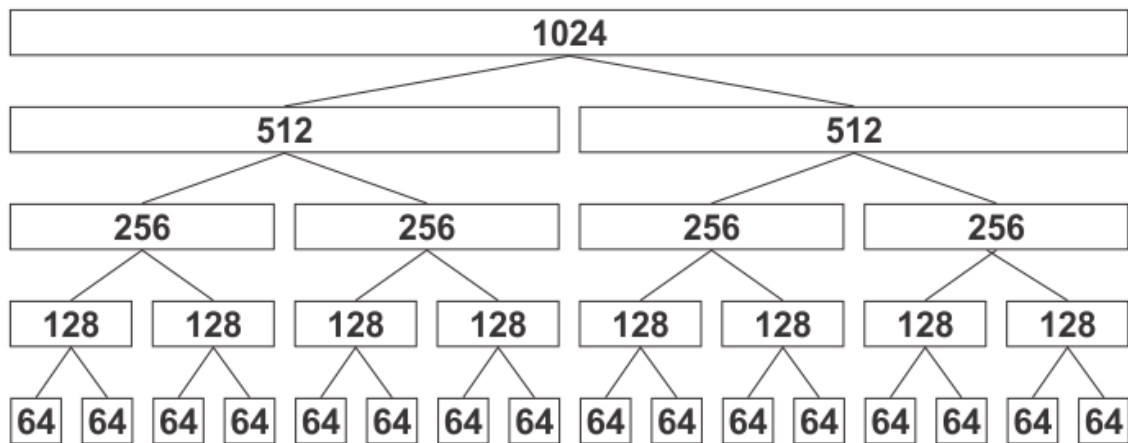
3. Второе разделение



Buddy allocator

Изначально единый блок свободной памяти также должен иметь размер, равный степени двойки, чтобы его разделение происходило в соответствии с заданными правилами

Buddy-пары для памяти 1024 байта:



Buddy allocator (применение)

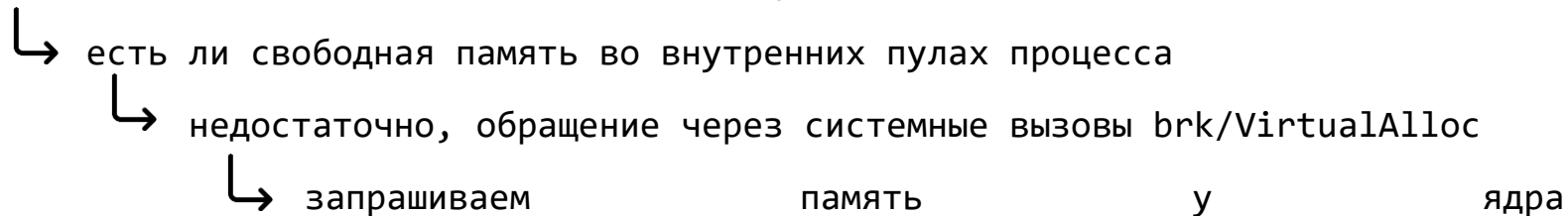
Отлично показал себя в игровых AI системах

- **Behavior tree**
деревья поведения с локальными блоками данных
- **State machine**
конечные автоматы с локальными состояниями
- **Pathfinding**
агенты навигации с “упакованными по степени двойки” размерами объектов
- **Decision trees**
деревья решений на основе нечеткой логики с локальными данными

Аллокатор с локальным кешем для потока

Системный `new/malloc` взаимодействует с операционной системой для выделения и освобождения памяти. Когда приложение запрашивает память, то в худшем случае ему приходится конкурировать за ресурсы с другими приложениями, работающими на той же машине.

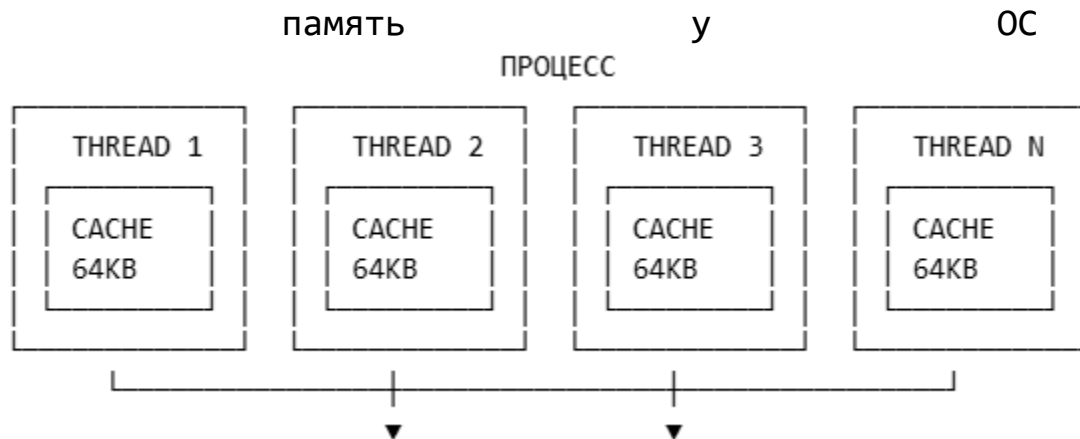
Взаимодействие происходит на нескольких уровнях:



Аллокатор с локальным кешем для потока

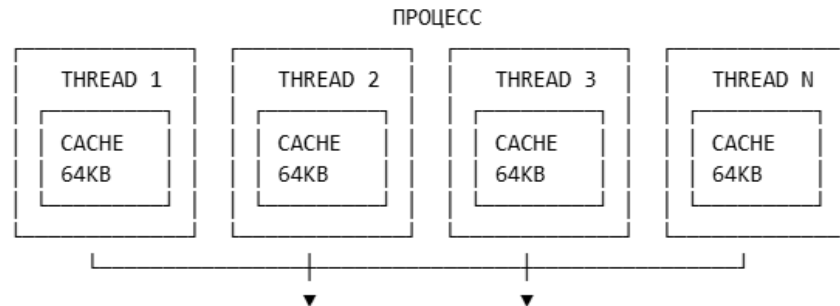
Меняем логику взаимодействия внутри процесса:

- ↳ есть ли свободная память во внутренних пулах треда
- ↳ недостаточно, запрашиваем память у общего пула приложения
- ↳ запрашиваем



Аллокатор с локальным кешем (особенности)

- **Lock-free**
каждый поток работает со своим локальным кешем без синхронизации
 - **Отсутствие**
потоки не конкурируют за общие ресурсы памяти
 - **Параллельное**
множественные потоки могут выделять память одновременно
 - **Масштабируемость**
производительность линейно растет с количеством потоков
- операции
contention
выделение



Фибоначчи аллокатор

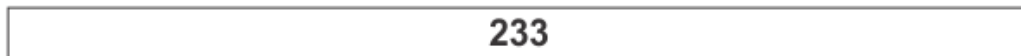
Аллокатор использует последовательность чисел Фибоначчи для управления размерами блоков памяти.

Числа Фибоначчи, применяемые к размеру выделяемого блока позволяют минимизировать фрагментацию и показывать лучшие результаты при работе со строками естественных языков и классами, которые не подвергаются искусственному уплотнению данных по степени двойки.

Фибоначчи аллокатор

Числа Фибоначчи, применяемые к размеру выделяемого блока позволяют минимизировать фрагментацию и показывать лучшие результаты при работе со строками естественных языков и классами.

1. Исходный блок (например, **233** байта)



2. Деление по Фибоначчи: **$233 = 144 + 89$**



3. Дальнейшее деление: **$144 = 89 + 55$**



4. Аллокация 20 байт → ближайший блок: 21 (**$55 = 34 + 21$**)



↑
Запрос
20 байт

↑
Ожидание запроса
~80 байт

Фибоначчи аллокатор

Показывает лучшие результаты при работе со строками естественных языков и классами.

Тип строки (символы)	2*-блок	Потери (байт)	F*-блок	Потери (байт)
13 (ключи)	16	3 (~23%)	13	0 (0%)
20 (имена, переменные)	32	12 (37%)	21	1 (~5%)
33 (метки времени)	64	31 (~48%)	34	1 (~2%)
50 (обычные строки)	64	14 (~22%)	55	5 (~9%)
80 (строки код)	128	48 (~37%)	89	9 (~10%)
90 (сообщения об ошибках)	128	32 (~30%)	144	54 (~38%)

Фибоначчи аллокатор (применение)

- **Системы с жесткими ограничениями памяти**
Встраиваемые устройства, микроконтроллеры
- **Там, где важна минимизация «дыр» из-за округления размера блока**
Парсеры естественных языков, компиляторы, интерпретаторы
- **Аллокаторы для специализированных пулов**
пул для объектов разного размера, но без жёсткой унификации по степени двойки

Какие аллокаторы использует Unity

- TLSF
- Разделяемый аллокатор для двух потоков
- Расширяемый аллокатор в куче
- Фиксированный аллокатор в куче
- Фиксированный аллокатор на стеке
- Линейный аллокатор
- Стековый аллокатор

Какие аллокаторы использует 4A Engine

- TLSF (общего назначения)
- Локальный аллокатор для потока (AI, render)
- Фиксированный аллокатор на стеке (оптимизации)
- Расширяемый аллокатор на стеке (оптимизации)
- Линейный аллокатор (render)

Какие аллокаторы использует Dagor

- Pool аллокатор (`dag::MemPool`)
- Frame аллокатор (`render`)
- Стековый, линейный аллокаторы (оптимизации)
- Heap allocator (`dag::MemAlloc`)
- Аллокатор для мелких объектов (оптимизации)

Какие аллокаторы использует Unreal

- Стековый аллокатор
- Линейный аллокатор
- Пул объектов (оптимизации)
- Аллокатор с разбиением на размеры

Какие аллокаторы использует CryEngine

- Обертка над системным malloc с выделением больших кусков памяти
- Pool аллокатор
- Buddy аллокатор (оптимизации)
- Фреймовый аллокатор
- Разные виды арена аллокаторов

Какие аллокаторы использует Sims 3

- Системный malloc/free разбитый на большие предвыделенные участки
- Pool аллокаторы для групп объектов
- Очень много статической памяти для разных подсистем
- Стековые аллокаторы
- Арена аллокаторы под разные размеры

Не были рассмотрены, но очень интересные

- Сжимающий аллокатор (compacting)
- TLSF (аллокатор с константным временем)
- Линейный аллокатор с откатом
- Отладочный цветной аллокатор (color-based)
- Отладочный аллокатор с фиксированием адресов
- Аллокатор с отслеживанием времени жизни (оптимизирующий)
- Отладочный (chaos) аллокатор
- Аллокатор с разделяемым кешем для потоков
- Локальный расширяемый аллокатор на стеке
- Локальный фиксированный аллокатор на стеке
- Аллокатор на стеке, который переживает стек
- Аллокатор готовых объектов
- Ареннный аллокатор объектов

Спасибо за внимание!
Приходите в мой блог на Хабре