Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Операционные системы»

Группа: М8О-215Б-23

Студент: Авраменко Д.А.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 27.12.24

Москва, 2024

Постановка задачи

Вариант 20

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.
В отчете необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов
- Процесс тестирования
- Обоснование подхода тестирования
- Результаты тестирования
- Заключение по проведенной работе

Задание

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и с алгоритмом Мак-Кьюзи-Кэрелса

Описание алгоритма: список свободных блоков (наиболее подходящее)

Вся память разделена на блоки. Каждый блок содержит служебную информацию (заголовок) и область данных. Блоки объединены в двусвязный список

При запросе памяти размера N:

1. Просматривается весь список свободных блоков
2. Среди всех подходящих блоков (размер ≥ N) выбирается блок с минимальным размером
3. Если найденный блок существенно больше требуемого, от него отделяется блок нужно размера
4. Выбранный блок помечается как занятый
5. Возвращается указатель на область данных

При освобождении блока:

1. Блок помечается как свободный
2. Проверяются соседние блоки
3. Если соседние блоки свободны, происходит слияние с ними

Описание алгоритма: Мак-Кьюзи-Кэрелса

Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Для управления страницами распределитель использует дополнительный массив kmemsizes[]. Каждая страница может находиться в одном из трех перечисленных состояний:

1. Быть свободной. В этом случае соответствующий элемент массива kmemsizes[] содержит указатель на элемент, описывающий следующую свободную страницу.
2. Быть разбитой на буферы определенного размера. Элемент массива содержит размер буфера.
3. Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в которой находятся данные о его длине.

Так как длина всех буферов одной страницы одинакова, нет нужды хранить в заголовках выделенных буферов указатель на список свободных буферов. Процедура free() находит страницу путем маскирования младших разрядов адреса буфера и обнаружения размера буфера в соответствующем элементе массива kmemsizes[]. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

Вызов процедуры malloc() заменен макроопределением, которое производит округление значения длины запрашиваемого участка вверх до достижения числа, являющегося степенью двойки (при этом не нужно прибавлять какие- либо дополнительные байты на заголовок) и удаляет буфер из соответствую- щего списка свободных буферов. Макрос вызывает функцию malloc() для за- проса одной или нескольких страниц тогда, когда список свободных буферов необходимого размера пуст. В этом случае malloc() вызывает процедуру, которая берет свободную страницу и разделяет ее на буферы необходимого размера.

Во время реализации алгоритма, было решено хранить список свободных страниц на первых страницах. Так как при хранении информации на страницах и при выделении данных, по размеру близких к размеру страницы, теряется много места

Некоторые листинги

list_of_free_blocks.cpp

```
void* alloc(size_t size) {
    if (size == 0) return nullptr;

    // Выравниваем размер и добавляем место под заголовок
    size_t aligned_size = (size + sizeof(BlockHeader) + 7) & ~7;

    // Ищем наиболее подходящий блок
    BlockHeader* best_block = nullptr;
    BlockHeader* current = head;
    size_t min_suitable_size = total_size + 1;

    while (current != nullptr) {
        if (current->is_free && current->size >= aligned_size) {
            if (current->size < min_suitable_size) {
                min_suitable_size = current->size;
                best_block = current;
            }
        }
        current = current->next;
    }

    if (!best_block) return nullptr;

    // Если блок достаточно большой, разделяем его
    if (best_block->size >= aligned_size + MINIMUM_BLOCK_SIZE) {
        BlockHeader* new_block = reinterpret_cast<BlockHeader*>(
            reinterpret_cast<char*>(best_block) + aligned_size
        );

        new_block->size = best_block->size - aligned_size;
```

```
            new_block->is_free = true;
            new_block->next = best_block->next;
            new_block->prev = best_block;

            if (best_block->next) {
                best_block->next->prev = new_block;
            }

            best_block->size = aligned_size;
            best_block->next = new_block;
        }

        best_block->is_free = false;
        return reinterpret_cast<char*>(best_block) + sizeof(BlockHeader);
    }

    void free(void* ptr) {
        if (!ptr) return;

        BlockHeader* block = reinterpret_cast<BlockHeader*>(
            static_cast<char*>(ptr) - sizeof(BlockHeader)
        );

        block->is_free = true;

        // Слияние с последующим свободным блоком
        if (block->next && block->next->is_free) {
            block->size += block->next->size;
            block->next = block->next->next;
            if (block->next) {
                block->next->prev = block;
            }
        }

        // Слияние с предыдущим свободным блоком
        if (block->prev && block->prev->is_free) {
            block->prev->size += block->size;
            block->prev->next = block->next;
            if (block->next) {
                block->next->prev = block->prev;
            }
        }
    }

};
```

Mckusey-Carels.cpp

```
void* alloc(size_t size) {
        if (size == 0) return nullptr;

        if (size <= MAX_BLOCK_SIZE) {
            // Выделение маленьких блоков
            size_t bucket = get_bucket_index(size);

            if (freelistarr[bucket] == nullptr) {
                if (free_page_info == nullptr) {
                    return nullptr;
                }

                // Разделение новой страницы на блоки
                PageInfo* page_info = free_page_info;
                free_page_info = free_page_info->next;
                split_page(page_info, MIN_BLOCK_SIZE << bucket);
            }

            // Получение блока из списка свободных
```

```
            FreeBlock* block = freelistarr[bucket];
            freelistarr[bucket] = block->next;
            return block;
        } else {
            // Выделение больших блоков
            size_t pages_needed = get_required_pages(size);

            PageInfo* start_page_info = find_consecutive_pages(pages_needed);
            if (!start_page_info) {
                return nullptr;
            }

            start_page_info->size = size;
            start_page_info->is_start_of_buffer = true;
            return start_page_info->page_addr;
        }
    }

    // Освобождение выделенной памяти
    void free(void* ptr) {
        if (!ptr) return;

        PageInfo* page_info = find_page_info(ptr);
        if (!page_info) return;

        if (page_info->size <= MAX_BLOCK_SIZE) {
            // Освобождение маленького блока
            size_t bucket = get_bucket_index(page_info->size);
            FreeBlock* block = static_cast<FreeBlock*>(ptr);
            block->next = freelistarr[bucket];
            freelistarr[bucket] = block;
        } else if (page_info->is_start_of_buffer) {
            // Освобождение большого блока
            size_t pages_count = get_required_pages(page_info->size);

            // Освобождение всех страниц в аллокации
            for (size_t i = 0; i < pages_count; ++i) {
                PageInfo* current = page_info + i;
                current->size = 0;
                current->is_start_of_buffer = false;
                insert_sorted(current);
            }

            coalesce_free_pages();
        }
    }
```

Подход к тестированию

Фактор использования

Формула расчета:

Фактор использования = (Полезная память / Общая выделенная память) * 100%

Где:

- Полезная память - память, реально используемая программой для данных
- Общая выделенная память - вся память, выделенная аллокатором, включая:
  - Полезную память
  - Служебные структуры (метаданные аллокатора)
  - Фрагментированные участки
  - Выравнивание
  - Накладные расходы

Было проведено три типа тестов:

- Выделение
- Выделение размеров, равных степени двойки (так как это является фишкой алгоритма Мак-Кьюзи-Кэрелса. Было сделано ради интереса

- Выделение и удаление с некоторым шансом

Время выделения и освобождения блоков

Для отслеживания времени будем использовать std::chrono::high_resolution_clock

Нужно будет провести так же несколько тестов:

- Тест на выделение маленьких размеров (меньше страницы)
- Тест на выделение больших размеров (больше страницы)
- Смешанный

Такое разделение нужно, чтобы протестировать выделение памяти на несколько страниц в алгоритме Мак-Кьюзи-Кэрелса

Ожидания и результаты тестирования

Для фактора использования с рандомными выделениями ожидаю безоговорочную победу для алгоритма с блоками, так как Мак-Кьюзи-Кэрелсу нужно дополнять до степени двойки. При степенях двойки жду победу Кэрелса.

Результаты:

McKusick-Karels:

Случайные размеры: 64.19%

Степени двойки: 91.62%

Случайные с освобождением: 62.26%

Best Fit:

Случайные размеры: 93.40%

Степени двойки: 88.65%

Случайные с освобождением: 90.81%

В итоге результаты совпали с ожиданием.

Для времени тестирования безусловно дольше должен работать алгоритм с блоками, так как ему для best fit нужно пройти весь список. Но на больших размерах и при большом количестве страниц Кэрелс должен работать медленнее себя же из-за поиска нескольких страниц подряд по нескольким страницам.

Результаты:

Running small blocks test...

=== Small Blocks Test ===

McKusick-Karels:

Alloc total: 0.001638912s

Alloc avg: 0.000000164s

Free total: 0.000263032s

Free avg: 0.000000026s

Best Fit:

Alloc total: 0.246580423s

Alloc avg: 0.000024658s

Free total: 0.000289679s

Free avg: 0.000000029s

Running large blocks test...

=== Large Blocks Test ===

McKusick-Karels:
  Alloc total: 0.000060893s
  Alloc avg: 0.000000061s
  Free total: 0.011228396s
  Free avg: 0.000011228s
Best Fit:
  Alloc total: 0.006717640s
  Alloc avg: 0.000006718s
  Free total: 0.000019078s
  Free avg: 0.000000019s

Running mixed sizes test...

=== Mixed Sizes Test ===
McKusick-Karels:
  Alloc total: 0.007649226s
  Alloc avg: 0.000000765s
  Free total: 0.043560325s
  Free avg: 0.000004356s
Best Fit:
  Alloc total: 0.470232250s
  Alloc avg: 0.000047023s
  Free total: 0.000311380s
  Free avg: 0.000000031s

Получили интересные результаты. На больших блоках выделение занимает мало времени, а освобождение – много. Скорее всего это из-за insert_sorted: insert_sorted проходит по списку для поиска правильной позиции и это происходит для каждой страницы в большом блоке.

Но зато с блоками результаты оказались предсказуемые.

Заключение

Списки блоков эффективнее, но медленнее. Если нужно хранить кратные двойки небольшие данные – Кэрелс, в остальных случаях – бест фит