МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Институт №8 "Информационные технологий и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

Курсовая работа
по курсу "Операционные системы"
3 семестр

Задание 19
Сравнение алгоритмов аллокации памяти

Студент: Леухин М. В.
Группа: М8О-206Б-20
Преподаватель: Соколов А. А.
Дата: 25.12.21
Оценка: 5
Подпись: _____

Москва, 2021

Содержание

1 Постановка задачи

Задание: Необходимо релизовать два алгоритма аллокации памяти и сравнить их. Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc. Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. В отчёте необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов.
- Процесс тестирования.
- Обоснование подхода тестирования.
- Результат тестирования.
- Заключение по проведённой работе.

Каждый аллокатор должен обладать следующим интерфейсом:

- Allocator* createMemoryAllocator(void* realMemory, size_t memory_size) - создание аллокатор памяти размера memory_size.
- void* alloc(Allocator* allocator, size_t block_size) - выделение памяти при помощи аллокатора размера block_size.
- void* free(Allocator* allocator, void* block) - возвращает выделенную память аллокатору.

Вариант 19: сравнить алгоритм аллокации на списках свободных блоков и на блоках по 2 в степени n.

2 Основная часть

2.1 Общая информация об аллокаторах

Аллокатор — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Стандартные функции malloc и free на самом деле имеют много проблем:

- Выделение нескольких байтов с помощью malloc происходит точно так же, как и выделение нескольких мегабайтов. В расчёт не берётся информация о том, что это за данные, где они будут распологаться и какой у них будет цикл жизни.
- Выделение памяти при помощи стандартных библиотечных функций или операторов обычно требует обращений к ядру операционной системы. Это может сказываться на производительности приложения.
- Они приводят к фрагментации кучи — состоянию, при котором информация в памяти разбросана в разных, не идущих последовательно блоках, из—за чего даже при достаточном суммарном объёме памяти возможна такая ситуация, что выделить блок в памяти для размещения информации будет невозможно.
- Плохая локальность указателей. Нет никакого способа узнать, какое именно место в память выделит вам malloc. Это может привести к тому, что будет происходить больше дорогостоящих промахов в кеше.

2.2 Аллокатор на списках свободных блоков

Суть аллокатора на списках свободных блоков заключается в том, что при каждом запросе на выделение памяти из доступной аллокатору памяти выделяется блока запрашиваемого размера (если это возможно). В конечном итоге вся память аллокатора будет разделена на список подряд идущих блоков, некоторые из которых будут знатяы, а некоторые свободны.

Вся память аллокатора разделена на блоки, каждый из которых содержит заголовок. В заголовке представлена информация о размере этого блока, размере предыдущего блока и логическая переменная, показывающая свободен ли блок. При создании такого аллокатора сразу выделяется память запрашиваемого размера, в которой создаётся один блок. При запросе на выделение памяти определённого размера ищется первый подходящий свободный блок, из которого выделяется блок запрашиваемого размера. При деаллокации ранее выделенной памяти сначала проверяется валидность переданного указателя — если пользователь хочет его вернуть аллокатору, значит этот указатель должен был быть когда-то этим же аллокатором выдан. После просто в заголовке меняем информацию о том, что блок доступен для использования.

Важной деталью этого аллокатора является возможность дефрагментации. При деаллокации определённого блока памяти происходит проверка доступности соседних блоков — если какой-то из этих блоков доступен, то происходит слияние текущего блока со свободным соседним.

Листинг аллокатора на свободных блоках:

```
class FreeBlocksAllocator : public Allocator {
public:

    explicit FreeBlocksAllocator(size_type size) {
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            return;
        }
        totalSize = size;
        endPointer = static_cast<void *>(static_cast<char
           *>(startPointer) + totalSize);
        auto *header = (Header *) startPointer;
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        header->previousSize = 0;
        usedSize = headerSize;
    };

    /**
     * Выделяетблокпамятизаданногоразмера      .
     * @param size - размерблокапамяти   .
     * @return указательнавыделенныйобъектпамятиилизначение
        null, еслипамятьнебылавыделена
     */
    pointer allocate(size_type size) override {
        if (size <= 0) {
            std::cerr << "Size must be bigger than 0\n";
            return nullptr;
        }
        if (size > totalSize - usedSize) { return nullptr; }
        auto *header = find(size);
        if (header == nullptr) { return nullptr; }
        splitBlock(header, size);
        return header + 1;
    };

    /**
     * Освобождаетпереданныйуказатель   .
     * @param ptr - указательдляосвобожденияпамяти    .
     */
    void deallocate(pointer ptr) override {
        if (!validateAddress(ptr)) {
            return;
        }
        auto *header = static_cast<Header *>(ptr) - 1;
        header->isAvailable = true;
        usedSize -= header->size;
        defragmentation(header);
    };
```

```
48
49 private:
50
51     /**
52      * Функцииипроверкидоступностиблизлежащихблоков      .
53      */
54     bool isPreviousFree(Header *header) {
55         auto *previous = header->previous();
56         return header != startPointer && previous->isAvailable;
57     }
58
59     bool isNextFree(Header *header) {
60         auto *next = header->next();
61         return header != endPointer && next->isAvailable;
62     }
63
64     /**
65      * Функцияслиянияисвободныхблоков     .
66      */
67     void defragmentation(Header *header) {
68         if (isPreviousFree(header)) {
69             auto *previous = header->previous();
70             if (header->next() != endPointer) {
71                 header->next()->previousSize += previous->size +
                       headerSize;
72             }
73             previous->size += header->size + headerSize;
74             usedSize -= headerSize;
75             header = previous;
76         }
77         if (isNextFree(header)) {
78             header->size += headerSize + header->next()->size;
79             usedSize -= headerSize;
80             auto *next = header->next();
81             if (next != endPointer) { next->previousSize =
                   header->size; }
82         }
83     }
84
85 };
```

Тестирование программы. Функция main:

```
1 int main(){
2
3     auto allocator = FreeBlocksAllocator(1024);
4     allocator.memoryDump();
5
6     auto ptr1 = allocator.allocate(200);
7     allocator.memoryDump();
8
```

```
    auto ptr2 = allocator.allocate(100);
    allocator.memoryDump();

    auto ptr3 = allocator.allocate(300);
    allocator.memoryDump();

    allocator.deallocate(ptr2);
    allocator.memoryDump();
    allocator.deallocate(ptr1);
    allocator.memoryDump();
    allocator.deallocate(ptr3);
    allocator.memoryDump();

}
```

Результат выполнения:

```
Total size: 1024
Used: 24
Header size: 24
+ 0x559b8ea269c0 1000

Total size: 1024
Used: 248
Header size: 24
- 0x559b8ea269c0 200
+ 0x559b8ea26aa0 776

Total size: 1024
Used: 372
Header size: 24
- 0x559b8ea269c0 200
- 0x559b8ea26aa0 100
+ 0x559b8ea26b1c 652

Total size: 1024
Used: 696
Header size: 24
- 0x559b8ea269c0 200
- 0x559b8ea26aa0 100
- 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

Total size: 1024
Used: 596
Header size: 24
- 0x559b8ea269c0 200
+ 0x559b8ea26aa0 100
- 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

```

```
Total size: 1024
Used: 372
Header size: 24
+ 0x559b8ea269c0 324
- 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

Total size: 1024
Used: 24
Header size: 24
+ 0x559b8ea269c0 1000


Process finished with exit code 0
```

2.3 Аллокатор на блоках размером 2^n

Принцип работы аллокатора на блоках размером 2^n аналогичен вышеописанному аллокатору на списках свободных блоков за тем исключением, что при создании аллокатора или при запросе на выделение памяти размер блоков выравнивается под ближайшую большую степень числа 2. Таким образом все используемые в данный момент блоки обязательно будут иметь размер 2^n. Такой алгоритм довольно близок к тому, что используется в стандартном системном аллокаторе, ведь он тоже оперирует с блоками, размер которых является степенью числа 2.

Листинг аллокатора на блоках размером 2^n:

```
class BinaryAllocator : public Allocator {
public:

    explicit BinaryAllocator(size_type size) {
        size = align(size);
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            return;
        }
        totalSize = size;
        endPointer = static_cast<void *>(static_cast<char
           *>(startPointer) + totalSize);
        auto *header = (Header *) startPointer;
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        header->previousSize = 0;
        usedSize = headerSize;
    };

    /**
     * Выравниваетразмерзапрашиваемойпамятидоближайшейстепени 2.
     */
```

```
22      static size_type align(size_type size) {
23          int i = 0;
24          while (pow(2, i) < size){
25              i++;
26          }
27          return (size_type) pow(2, i);
28      }
29
30      /**
31       * Выделяетблокпамятизаданногоразмера      .
32       * @param size – размерблокапамяти   .
33       * @return указательнавыделенныйобъектпамятиилизначение
             null, еслипамятьнебылавыделена
34       */
35      pointer allocate(size_type size) override {
36          if (size <= 0) {
37              std::cerr << "Size must be bigger than 0\n";
38              return nullptr;
39          }
40          size = align(size);
41          if (size > totalSize - usedSize) { return nullptr; }
42          auto *header = find(size);
43          if (header == nullptr) { return nullptr; }
44          splitBlock(header, size);
45          return header + 1;
46      };
47
48      /**
49       * Освобождаетпереданныйуказатель   .
50       * @param ptr – указательдляосвобожденияпамяти    .
51       */
52      void deallocate(pointer ptr) override {
53          if (!validateAddress(ptr)) {
54              return;
55          }
56          auto *header = static_cast<Header *>(ptr) - 1;
57          header->isAvailable = true;
58          usedSize -= header->size;
59          defragmentation(header);
60      };
61
62  private:
63
64      /**
65       * Функциипроверкидоступностиблизлежащихблоков      .
66       */
67      bool isPreviousFree(Header *header) {
68          auto *previous = header->previous();
69          return header != startPointer && previous->isAvailable;
70      }
```

```
71
72      bool isNextFree(Header *header) {
73          auto *next = header->next();
74          return header != endPointer && next->isAvailable;
75      }
76
77      /**
78       * Функцияслияниясвободныхблоков     .
79       */
80      void defragmentation(Header *header) {
81          if (isPreviousFree(header)) {
82              auto *previous = header->previous();
83              if (header->next() < endPointer) {
84                  header->next()->previousSize += previous->size +
                        headerSize;
85              }
86              previous->size += header->size + headerSize;
87              usedSize -= headerSize;
88              header = previous;
89          }
90          if (isNextFree(header)) {
91              header->size += headerSize + header->next()->size;
92              usedSize -= headerSize;
93              auto *next = header->next();
94              if (next != endPointer) { next->previousSize =
                    header->size; }
95          }
96      }
97
98  };
```

Тестирование программы. Функция main:

```
1   int main(){
2
3       auto allocator = BinaryAllocator(1024);
4       allocator.memoryDump();
5
6       auto ptr1 = allocator.allocate(200);
7       allocator.memoryDump();
8
9       auto ptr2 = allocator.allocate(100);
10      allocator.memoryDump();
11
12      auto ptr3 = allocator.allocate(300);
13      allocator.memoryDump();
14
15      allocator.deallocate(ptr2);
16      allocator.memoryDump();
17      allocator.deallocate(ptr1);
18      allocator.memoryDump();
```

```
19      allocator.deallocate(ptr3);
20      allocator.memoryDump();
21
22  }
```

Результат тестирования:

```
1  Total size: 1024
2  Used: 24
3  Header size: 24
4  + 0x55c4a420d9c0 1000
5
6  Total size: 1024
7  Used: 304
8  Header size: 24
9  - 0x55c4a420d9c0 256
10 + 0x55c4a420dad8 720
11
12 Total size: 1024
13 Used: 456
14 Header size: 24
15 - 0x55c4a420d9c0 256
16 - 0x55c4a420dad8 128
17 + 0x55c4a420db70 568
18
19 Total size: 1024
20 Used: 992
21 Header size: 24
22 - 0x55c4a420d9c0 256
23 - 0x55c4a420dad8 128
24 - 0x55c4a420db70 512
25 + 0x55c4a420dd88 32
26
27 Total size: 1024
28 Used: 864
29 Header size: 24
30 - 0x55c4a420d9c0 256
31 + 0x55c4a420dad8 128
32 - 0x55c4a420db70 512
33 + 0x55c4a420dd88 32
34
35 Total size: 1024
36 Used: 584
37 Header size: 24
38 + 0x55c4a420d9c0 408
39 - 0x55c4a420db70 512
40 + 0x55c4a420dd88 32
41
42 Total size: 1024
43 Used: 24
44 Header size: 24
```

```
+ 0x55c4a420d9c0 1000


Process finished with exit code 0
```

2.4 Сравнение алгоритмов аллокации

Для сравнения алогоритмов аллокации будем замерять общее время работы аллокатора по аллокации / деаллокации блоков памяти. Для того, чтобы значения были более наглядными, тестовый файл содержит 50000 запросов на аллокацию / деаллокацию. На основе тестирования были получены следующие показатели:

- Аллокатор на списке свободных блоков с дефрагментацией: 196.185 мс, 184.035 мс, 193.589 мс.
- Аллокатор на списке свободных блоков без дефрагментации: 317.919 мс, 320.539 мс, 325.483 мс.
- Аллокатор на блоках по 2^n с дефрагментацией: 155.929 мс, 163.766 мс, 166.549 мс.
- Аллокатор на блоках по 2^n без дефрагментации: 129.728 мс, 126.534 мс, 132.30 мс.

3 Вывод

По результатам тестирования видно, что самый худший результат показывает алгоритм аллокации на списках свободных блоков без дефрагментации, а самый лучший - аллокатор на блоках по 2^n, причём так же без дефрагментации. Причиной этого является то, что процессор и вся система памяти изначально настроена на работу в бинарной системе, то есть с блоками памяти, размер которых является степенью числа 2. При аллокации блока памяти, размер которого отличается от степени 2, система всё равно приводит его к такому виду. В случае с аллокатором на блоках по 2^n мы сразу работаем с "более удобными"для процессора блоками. Причина того, почему лучше всего работает алгоритм без дефрагментации также проста — при выбранной реализации аллокатора в общем случае невозможно провести дефрагментацию двух соседних блоков памяти так, чтобы размер полученного блока памяти всё равно был степенью числа 2 из-за того, что вместе с каждым блоком памяти хранится его заголовок, который при слиянии становится доступной памятью. То есть при слиянии двух блоков размер полученного блока равен $s = 2^{n_1} + 2^{n_2} + 24$, где 2^{n_1} и 2^{n_2} — размеры соседних блоков в байтах, а 24 — размер заголовка. В общем случае s не будет являться степенью числа 2 (хоть это и возможно, например, если $n_1 = n_2 = 2$, так как в таком случае $4+4+24 = 32 = 2^5$).