Санкт-Петербургский политехнический университет

Высшая школа прикладной математики и вычислительной физики, ИПММ

Направление подготовки

«01.03.02 Прикладная математика и информатика»

Курсовая работа

Тема "Управление памятью. Следующий подходящий (next fit strategy). Односвязный список"

Дисциплина "Алгоритмы и базы данных"

Выполнил студент гр. **3630102/90002** Ушков В. А.

Преподаватель: Беляев С. Ю.

Санкт-Петербург

2020

Условие задачи

Реализуйте систему управления памятью, построенную на списках пустых блоков. Используйте односвязный список и стратегию “Следующий подходящий”.

Напишите функции выделения и освобождения памяти блоками произвольного размера.

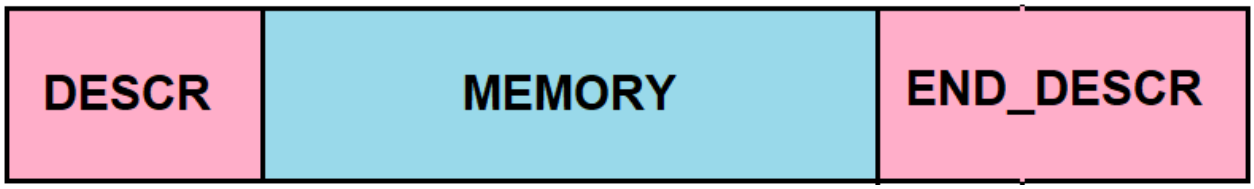
Решение задачи

Каждый блок памяти обособляется структурами **DESCR** и **END_DESCR** для возможности дефрагментации

```
typedef struct descriptor_t {
    int size;
    int key;
    struct descriptor_t* next;
} DESCR;

typedef struct end_descr_t {
    int size;
} END_DESCR;
```

Таким образом, блоки представляются в виде



Информация о состоянии системы хранится в глобальной структуре **memory_system**

```
struct {
    DESCR* list;
    char* start;
    int total_size;
} memory_system = { (DESCR*)NULL, 0, NULL };
```

Обрабатываются следующие ситуации

- При выделении памяти
    - Подходящего блока нет – система не имеет возможности выделить память, поэтому возвращает нулевой указатель
    - Подходящий блок невозможно разделить надвое так, чтобы первый оставался подходящим, а второй был ненулевым – блок удаляется из списка свободного пространства, возвращается указатель на этот блок
    - Подходящий блок можно разделить надвое – блок удаляется из списка свободного пространства, делится на две части,

возвращаем указатель на первый блок, а второй заносим в список свободного пространства

- При освобождении памяти
  - Блок уже не занят – освобождение не требуется
  - Блок требует объединения слева
  - Блок требует объединения справа
  - Блок требует объединения и слева, и справа – для последних трёх случаев проводим дефрагментацию: удаляем все объединяемые блоки, обнуляем поля key (кроме левого блока), отмечаем новый размер блока в левый DESCR и правый END_DESCR и добавляем получившийся блок в список свободного пространства
  - Блок не требует объединения – блок освобождается без дефрагментации и заносится в список свободного пространства

Затраты ресурсов

И для выделения, и для освобождения памяти необходимо Θ(1) памяти, временные затраты при выделении пропорциональны затратам на поиск подходящего элемента. При освобождении памяти временные затраты также пропорциональны поиску следующего блока в списке, если блок необходимо объединять, иначе временная сложность равна Θ(1)

Альтернативные алгоритмы

Помимо стратегии «Следующий подходящий» существуют стратегии «Первый подходящий» (first fit, когда из всех блоков выбирается первый достаточный для запроса), а также «Лучший подходящий» (best fit, выбирается наименьший подходящий блок)

Также возможна реализация на двусвязном списке, которая будет более выгодна, так как освобождение блока при использовании двусвязного списка всегда занимает Θ(1) времени, тогда как для односвязного эта величина может быть пропорциональна временной сложности поиска следующего блока в списке

Листинг кода

```c
#include "memallocator.h"
#include <stdio.h>
#include <stdlib.h>

#define FAIL 0
#define BLOCK_USED -1
#define MAGICKEY 3131

typedef enum {
    NO_FIT,
    FIRST_FIT,
    NEXT_FIT,
}FIT;

typedef struct descriptor_t {
    int size;
    int key;
    struct descriptor_t* next;
} DESCR;

typedef struct end_descr_t {
    int size;
} END_DESCR;

struct {
    DESCR* list;
    char* start;
    int total_size;
} memory_system = { (DESCR*)NULL, 0, NULL };

int memgetminimumsize() {
    return (int)(sizeof(DESCR) + sizeof(END_DESCR));
}
```

```c
int memgetblocksize() {
    return (int)(sizeof(DESCR) + sizeof(END_DESCR));
}

int meminit(void* pMemory, int size) {
    int min = memgetminimumsize();
    if (!pMemory || size < min)
        return FAIL;
    else {
        memory_system.total_size = size;
        memory_system.start = (char*)pMemory;
        memory_system.list = (DESCR*)pMemory;
        memory_system.list->key = MAGICKEY;
        memory_system.list->size = size - min;
        memory_system.list->next = NULL;
        ((END_DESCR*)(memory_system.start + sizeof(DESCR) + memory_system.list->size))->size = memory_system.list->size;
        return SUCCESS;
    }
}

void* memalloc(int size) {
    if (!memory_system.list || size <= 0 || size + memgetminimumsize() > memory_system.total_size)
        return NULL;

    DESCR** temp = &memory_system.list;
    DESCR** current = NULL;
    DESCR** first_fit = NULL;
    DESCR* new_descr = NULL;
    FIT fit = NO_FIT;

    while (*temp != NULL) {
        if ((*temp)->size >= size) {
            if (fit == NO_FIT) {
```

```
            fit = FIRST_FIT;
            first_fit = temp;
        }
        if (fit == FIRST_FIT) {
            fit = NEXT_FIT;
            break;
        }
    }
    current = temp;
    temp = &(*temp)->next;
}

if (fit == NO_FIT)
    return NULL;
else {
    if (fit == FIRST_FIT) {
        current = first_fit;
    }
    if (fit == NEXT_FIT)
        current = temp;

    new_descr = (*current);
    new_descr->key = MAGICKEY;

    int prev_size = (*current)->size;

    if (prev_size <= size + memgetblocksize()) {
        (*current) = (*current)->next;
        ((END_DESCR*)((char*)new_descr + sizeof(DESCR) + new_descr->size))->size = BLOCK_USED;
        new_descr->next = NULL;
        return (void*)((char*)new_descr + sizeof(DESCR));
    }
    else {
        (*current) = (DESCR*)((char*)new_descr + memgetblocksize() + size);
```

```
        (*current)->size = new_descr->size - memgetblocksize() - size;
        (*current)->key = MAGICKEY;
        (*current)->next = new_descr->next;
        ((END_DESCR*)((char*)new_descr + sizeof(DESCR) + new_descr->size))->size = (*current)->size;

        new_descr->size = size;
        new_descr->next = NULL;
        ((END_DESCR*)((char*)new_descr + sizeof(DESCR) + new_descr->size))->size = BLOCK_USED;

        return (void*)((char*)new_descr + sizeof(DESCR));
    }
  }
}

void memfree(void* p) {
  if ((char*)p < (char*)memory_system.start || (char*)p >= (char*)memory_system.start + memory_system.total_size || p == NULL)
      return;
  if (memory_system.total_size == 4096)
      return;
  DESCR* current = (DESCR*)((char*)p - sizeof(DESCR));
  END_DESCR* end_current = (END_DESCR*)((char*)p + current->size);

  if (current->key != MAGICKEY)
      return;

  END_DESCR* end_left = (END_DESCR*)((char*)current - sizeof(END_DESCR));
  DESCR* left_descr = (DESCR*)((char*)end_left - end_left->size - sizeof(DESCR));

  if ((char*)end_left < (char*)memory_system.start || end_left->size == BLOCK_USED) {
      current->next = memory_system.list;
      end_current->size = current->size;
      memory_system.list = current;
  }
  else {
```

```
        left_descr->size += current->size + memgetblocksize();
        end_current->size = left_descr->size;
        current = left_descr;
    }

    DESCR* right_descr = (DESCR*)((char*)end_current + sizeof(END_DESCR));
    END_DESCR* end_right = (END_DESCR*)((char*)right_descr + right_descr->size + sizeof(DESCR));

    if ((char*)right_descr >= (char*)memory_system.start + memory_system.total_size || right_descr->key != MAGICKEY)
        return;
    if (end_right->size != BLOCK_USED) {
        current->size += right_descr->size + memgetblocksize();
        end_right->size = current->size;
        right_descr->key = 0;
        DESCR* temp = memory_system.list;
        while (temp != right_descr && temp != NULL) {
            temp = temp->next;
        }
        if (temp == right_descr)
            temp = temp->next;
    }
}

void memdone() {
    memory_system.total_size = 0;
    memory_system.start = NULL;
    memory_system.list = NULL;
}
```

Выводы

Стратегия «Следующий подходящий» является довольно выгодным алгоритмом, так как работает почти так же быстро как «Первый подходящий» и быстрее, чем «Лучший подходящий», при этом память будет распределяться более-менее равномерно, в отличии от «ПП»