Отчет

по лабораторной работе № 4

«Управление динамическим выделением памяти»

по дисциплине

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Выполнил студент гр. ИС/б-22о

Горбенко К.Н.

Проверила:

Шалимова Е.М.

**Цель работы**: изучение алгоритмов выделения памяти и программирование основных операций, выполняемых с линейными списковыми структурами.

* 1. Постановка задачи

Разработать подсистему динамического выделения памяти, язык программирования С. Для отладки разработать программу, в которой предусмотреть формирование списков свободной и выделенной памяти, имитацию запроса и выполнения заданной операции. Элемент списка должен содержать адрес первого байта блока памяти и размер блока. В случае, когда освобождающийся блок примыкает к соседнему блоку, он должен объединяться с ним в один блок. В программе обеспечить вывод исходных данных и результатов.

**Вариант № 4:**

* стратегия выделения памяти: первый подходящий;
* метод сортировки: сортировка обменом;
* тип списка: двунаправленный;
  1. Описание логики работы программы

Программа начинается со ввода исходного состояния памяти из файла в массив. Затем из этого массива формируются два списка: список выделенной памяти и список свободной памяти. После этого вызывается меню, содержащее следующие пункты:

1. показать текущее состояние памяти;
2. выделить память;
3. освободить память;
4. показать текущий список свободной памяти;
5. показать текущий список выделенной памяти;
6. выход.

Показать текущее состояние памяти включает в себя вывод массива с текущим состоянием на экран.

Выделить память включает в себя ввод количества байт для выделения, извлечение подходящего блока из списка свободной памяти и помещение его в список выделенной памяти. Затем происходит обновление состояния массива, содержащего текущее состояние памяти, и сортировка списка выделенной памяти.

Освободить память включает в себя ввод количества байт для освобождения, извлечение подходящего блока из списка выделенной памяти и помещение его в список свободной памяти. При поиске подходящего блока для освобождения учитываются только те блоки, длина которых равна введенному числу байт. Затем происходит обновление состояния массива, содержащего текущее состояние памяти, сортировка списка свободной памяти, проверка вставленного блока памяти на возможность объединения с другими блоками.

Пункты 4 и 5 включают в себя вывод соответствующих списков на экран.

* 1. Описание используемых структур данных

В программе используется статический массив, содержащий текущее состояние памяти, а также структура, представляющая элемент списка блока памяти. Целью использования данного массива является возможность наглядно увидеть текущее состояние памяти. Структура включает в себя данные о позиции начала блока, длине блока и указатели на предыдущий и следующий элементы списка:

struct MemoryNode

{

int startPosition;

int bytes;

MemoryNode \* next;

MemoryNode \* previous;

};

* 1. Спецификации подпрограмм

void show\_current\_state(int \* currentState)

Функция принимает указатель на первый элемент массива, содержащего текущее состояние памяти и выводит этот массив на экран.

int get\_user\_option()

Функция выводит меню на экран и возвращает номер выбранного пользователем пункта меню.

MemoryNode \* get\_memory\_list(int \* state, int length, int number)

Функция принимает указатель на первый элемент массива, содержащего текущее состояние памяти, его длину и число, представляющее вид памяти (0 – свободная, 1 – выделенная) и возвращает указатель на первый элемент сформированного из массива списка соответствующей памяти.

void show\_list(MemoryNode \* list)

Функция принимает указатель на первый элемент списка любой памяти и выводит этот список на экран.

void bubbleSort(MemoryNode \*start)

Функция принимает указатель на первый элемент списка памяти и сортирует этот список методом прямого обмена. Обмениваются не элементы списка, а только их значения.

int remove\_by\_bytes(MemoryNode \*& list, int bytes)

Функция принимает указатель на первый элемент списка по ссылке и количество байт для извлечения подходящего блока памяти и возвращает позицию его начала. Если подходящий блок не найден, функция вернет -1.

void push\_to\_end(MemoryNode \* list, int bytes, int start)

Функция принимает указатель на первый элемент списка, число байт и число первой позиции блока для помещения соответствующей записи в конец данного списка.

void change\_memory\_state(int startPosition, int bytes, int\* currentState)

Функция принимает число байт, начальную позицию блока и указатель на первый элемент массива, содержащего текущее состояние памяти и обновляет его с учетом изменившихся данных блока.

void check\_for\_collitions(MemoryNode\* list, int startPosition)

Функция принимает указатель на первый элемент списка и позицию изменившегося блока для проверки этого блока на возможность склеивания с соседними.

void freeMemory(MemoryNode\* list)

Функция принимает указатель на первый элемент списка памяти и освобождает динамически выделенную под его элементы память.

* 1. Результат работы програмы

Запустим программу и выведем начальное состояние памяти на экран:

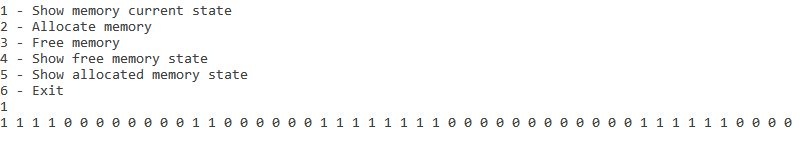


Рис. 1 – Начальное состояние памяти

Выделим 4 байта памяти. Для выделения должен использоваться первый подходящий блок памяти.

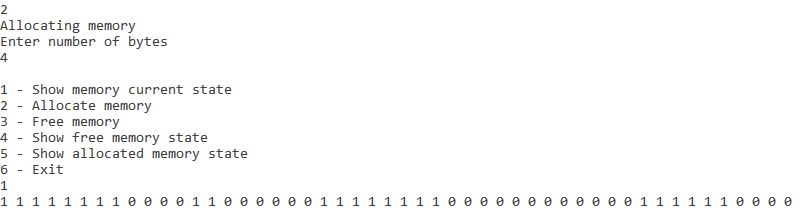


Рис. 2 – Память после выделения 4 байт

Освободим память из блока, содержащего 8 байт. При этом элементы списка свободной памяти должны склеиться между собой:

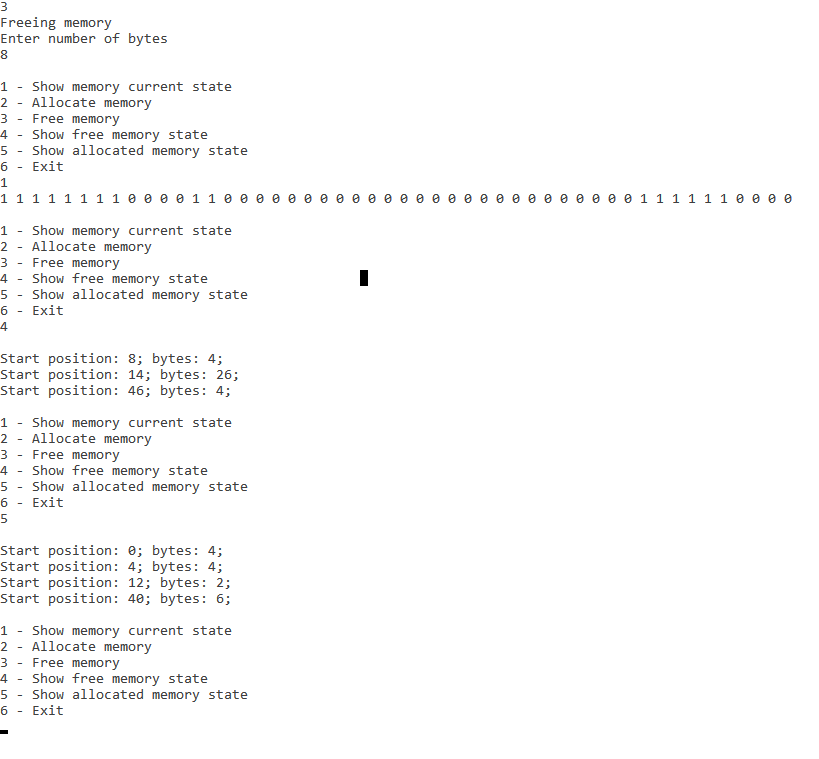


Рис. 3 – Память после освобождения 8 байт

Судя по списку свободной памяти, элементы списка свободной памяти склеились между собой.

Проверим, что программа не даст освободить память из блока, длина которого не равна введенному числу байт:

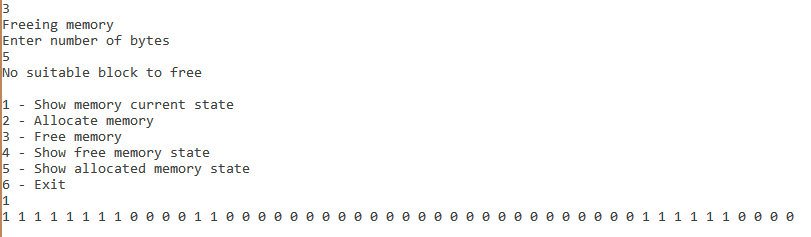


Рис. 4 – Память после попытки освободить несуществующий блок

Как видно из рисунка № 4, блок памяти длиной 5 байт не был освобожден так как в списке не было соответствующего блока.

* 1. Текст программы

#include "pch.h"

#include <iostream>

#include <fstream>

#include <stdio.h>

using namespace std;

struct MemoryNode

{

int startPosition;

int bytes;

MemoryNode \* next;

MemoryNode \* previous;

};

void show\_current\_state(int \* currentState)

{

for (int i = 0; i < 50; i++)

cout << currentState[i] << " ";

cout << endl;

}

int get\_user\_option()

{

int userOption;

cout << endl;

cout << "1 - Show memory current state" << endl;

cout << "2 - Allocate memory" << endl;

cout << "3 - Free memory" << endl;

cout << "4 - Show free memory state" << endl;

cout << "5 - Show allocated memory state" << endl;

cout << "6 - Exit" << endl;

cin >> userOption;

return userOption;

}

void set\_current\_node\_state(MemoryNode \* current, MemoryNode\* previous, int i, int counter)

{

current->startPosition = i - counter;

current->bytes = counter;

current->previous = previous;

}

MemoryNode \* get\_memory\_list(int \* state, int length, int number)

{

MemoryNode \* first = (MemoryNode\*)malloc(sizeof(MemoryNode));

MemoryNode \* current = first;

MemoryNode \* previous = first;

int counter = 0;

for (int i = 0; i < length; i++)

{

if (current == NULL && state[i] == number)

{

current = (MemoryNode\*)malloc(sizeof(MemoryNode));

previous->next = current;

}

if (state[i] == number)

counter++;

else if (counter > 0)

{

set\_current\_node\_state(current, previous, i, counter);

previous = current;

current->next = NULL;

current = current->next;

counter = 0;

}

}

if (counter > 0)

{

set\_current\_node\_state(current, previous, 50, counter);

previous = current;

current->next = NULL;

current = current->next;

}

first->previous = NULL;

return first;

}

void show\_list(MemoryNode \* list)

{

MemoryNode \* node = list;

cout << endl;

while (node != NULL)

{

cout << "Start position: " << node->startPosition << "; bytes: " << node->bytes << ";" << endl;

node = node->next;

}

}

void swap(MemoryNode \*node, MemoryNode \*next)

{

int tempBytes = node->bytes;

int tempStart = node->startPosition;

node->bytes = next->bytes;

node->startPosition = next->startPosition;

next->bytes = tempBytes;

next->startPosition = tempStart;

}

void bubbleSort(MemoryNode \*start)

{

int swapped;

MemoryNode \*node;

MemoryNode \*next;

if (start == NULL) return;

do

{

swapped = 0;

node = start;

while (node->next != NULL)

{

next = node->next;

if (node->startPosition > next->startPosition)

{

swap(node, next);

swapped = 1;

}

node = node->next;

}

} while (swapped);

}

int remove\_by\_bytes(MemoryNode \*& list, int bytes)

{

int startPosition;

MemoryNode\* current = list;

MemoryNode\* suitable = NULL;

while (current->next != NULL && suitable == NULL)

{

if (current->bytes >= bytes)

suitable = current;

current = current->next;

}

if (suitable == NULL)

return -1;

startPosition = suitable->startPosition;

if (suitable->bytes > bytes)

{

suitable->bytes = suitable->bytes - bytes;

suitable->startPosition = suitable->startPosition + bytes;

}

else

{

if (suitable->previous == NULL && suitable->next == NULL){

list = NULL;

}

else if (suitable->previous == NULL) {

list = suitable->next;

suitable->next->previous = NULL;

}

else if (suitable->next == NULL) {

suitable->previous->next = NULL;

}

else

{

suitable->previous->next = suitable->next;

suitable->next->previous = suitable->previous;

}

free(suitable);

}

return startPosition;

}

int remove\_by\_bytes\_strictly(MemoryNode \*& list, int bytes)

{

int startPosition;

MemoryNode\* current = list;

MemoryNode\* suitable = NULL;

while (current->next != NULL && suitable == NULL)

{

if (current->bytes == bytes)

suitable = current;

current = current->next;

}

if (suitable == NULL)

return -1;

startPosition = suitable->startPosition;

if (suitable->previous == NULL && suitable->next == NULL) {

list = NULL;

}

else if (suitable->previous == NULL) {

list = suitable->next;

suitable->next->previous = NULL;

}

else if (suitable->next == NULL) {

suitable->previous->next = NULL;

}

else

{

suitable->previous->next = suitable->next;

suitable->next->previous = suitable->previous;

}

free(suitable);

return startPosition;

}

void push\_to\_end(MemoryNode \* list, int bytes, int start) {

MemoryNode\* current = list;

while (current->next != NULL) {

current = current->next;

}

current->next = (MemoryNode\*)malloc(sizeof(MemoryNode));

current->next->bytes = bytes;

current->next->startPosition = start;

current->next->next = NULL;

current->next->previous = current;

}

void change\_memory\_state(int startPosition, int bytes, int\* currentState)

{

int state = currentState[startPosition];

if (state == 0) state = 1;

else state = 0;

for (int i = startPosition; i < startPosition + bytes; i++)

currentState[i] = state;

}

void check\_for\_collitions(MemoryNode\* list, int startPosition)

{

MemoryNode\* node = NULL;

MemoryNode\* previous = NULL;

MemoryNode\* next = NULL;

while (list->next != NULL && node == NULL) {

if (list->startPosition == startPosition)

node = list;

list = list->next;

}

if (node->previous == NULL && node->next == NULL)

return;

if (node->previous != NULL)

{

previous = node->previous;

if (previous->startPosition + previous->bytes == node->startPosition)

{

previous->bytes += node->bytes;

previous->next = node->next;

free(node);

node = previous;

}

}

previous = node->previous;

if (node->next != NULL)

{

next = node->next;

if (node->startPosition + node->bytes == next->startPosition)

{

node->bytes += next->bytes;

node->next = next->next;

if (next->next != NULL)

next->next->previous = node;

free(next);

}

}

}

void freeMemory(MemoryNode\* list)

{

MemoryNode\* current = list;

MemoryNode\* next;

while (current->next != NULL)

{

next = current->next;

free(current);

current = next;

}

free(current);

}

int main()

{

ifstream myFile("D:\\Learning\\ОС\\4 семестр\\Лабораторные работы\\ЛР4\\Lab4\\Debug\\file.txt");

int currentState[50];

for (int i = 0; i < 50; i++)

myFile >> currentState[i];

MemoryNode \* free = get\_memory\_list(currentState, 50, 0);

MemoryNode \* allocated = get\_memory\_list(currentState, 50, 1);

int userOption;

int bytes;

int startPosition;

do

{

userOption = get\_user\_option();

switch (userOption)

{

case 1:

show\_current\_state(currentState);

break;

case 2:

cout << "Allocating memory" << endl;

cout << "Enter number of bytes" << endl;

cin >> bytes;

startPosition = remove\_by\_bytes(free, bytes);

if (startPosition != -1) {

push\_to\_end(allocated, bytes, startPosition);

change\_memory\_state(startPosition, bytes, currentState);

bubbleSort(allocated);

}

break;

case 3:

cout << "Freeing memory" << endl;

cout << "Enter number of bytes" << endl;

cin >> bytes;

startPosition = remove\_by\_bytes\_strictly(allocated, bytes);

if (startPosition != -1){

push\_to\_end(free, bytes, startPosition);

change\_memory\_state(startPosition, bytes, currentState);

bubbleSort(free);

check\_for\_collitions(free, startPosition);

}

else cout << "No suitable block to free" << endl;

break;

case 4:

show\_list(free);

break;

case 5:

show\_list(allocated);

break;

case 6:

freeMemory(free);

freeMemory(allocated);

return 0;

}

} while (true);

}

* 1. Вывод

В ходе лабораторной работы были изучены алгоритмы выделения памяти в операционных системах. Для данной задачи используются два связных списка, представляющих блоки свободной и выделенной памяти. При выделении или освобождении памяти соответствующий блок перемещается в противоположный список. При освобождении памяти возможна склейка последовательных блоков свободной памяти.