**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9**

по дисциплине «Операционные системы»

на тему: «Простейшие схемы управления памятью»

Выполнил: студент гр. ИТИ-11

Филончик К.В.

Принял: преподаватель Карась О.В.

Гомель 2022

**Цель работы:** изучить алгоритмы управления памятью, разработать программу менеджера памяти.

**Теоретические сведения:**

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера. Смежные свободные участки могут быть объединены.

Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память) В какой раздел помещать процесс? Наиболее распространены три стратегии:

− стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел;

− стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места;

− стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

**Задание:**

Разработать программу, реализующую заданный алгоритм выделения памяти. Менеджер памяти должен:

1. По запросу процесса выделять память, согласно заданного алгоритма (таблица). На экран должна выводиться следующая информация о состоянии памяти: объем памяти, объѐм свободной памяти, размер наибольшего свободного блока, количество запросов на выделение памяти, количество удовлетворѐнных запросов (%).

2. Для выделения памяти указывается имя процесса и размер блока. После нажатия на кнопку «ДОБАВИТЬ» память выделяется или выдаѐтся сообщение о невозможности выделения.

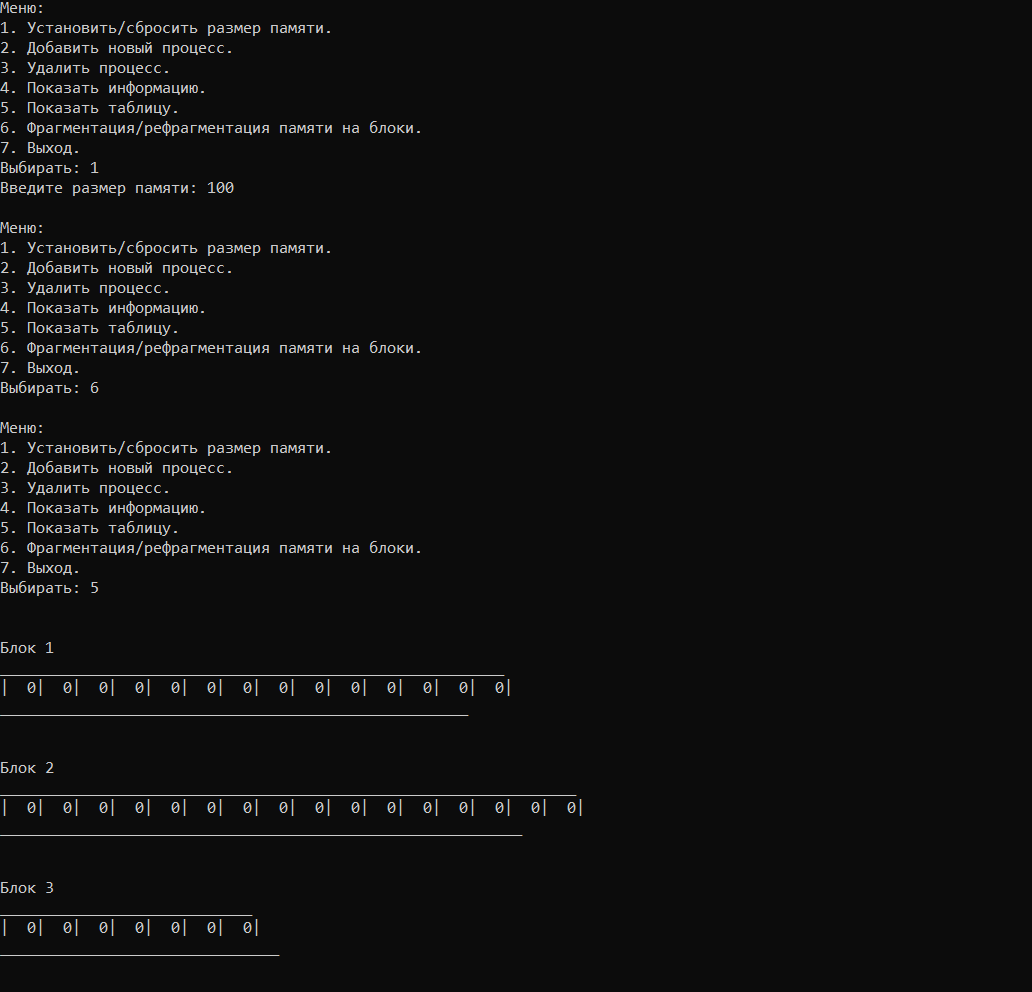
3. Удалять из памяти заданный блок или все блоки заданного процесса (по нажатию кнопки «УДАЛИТЬ»). Указывается номер удаляемого блока и имя процесса.

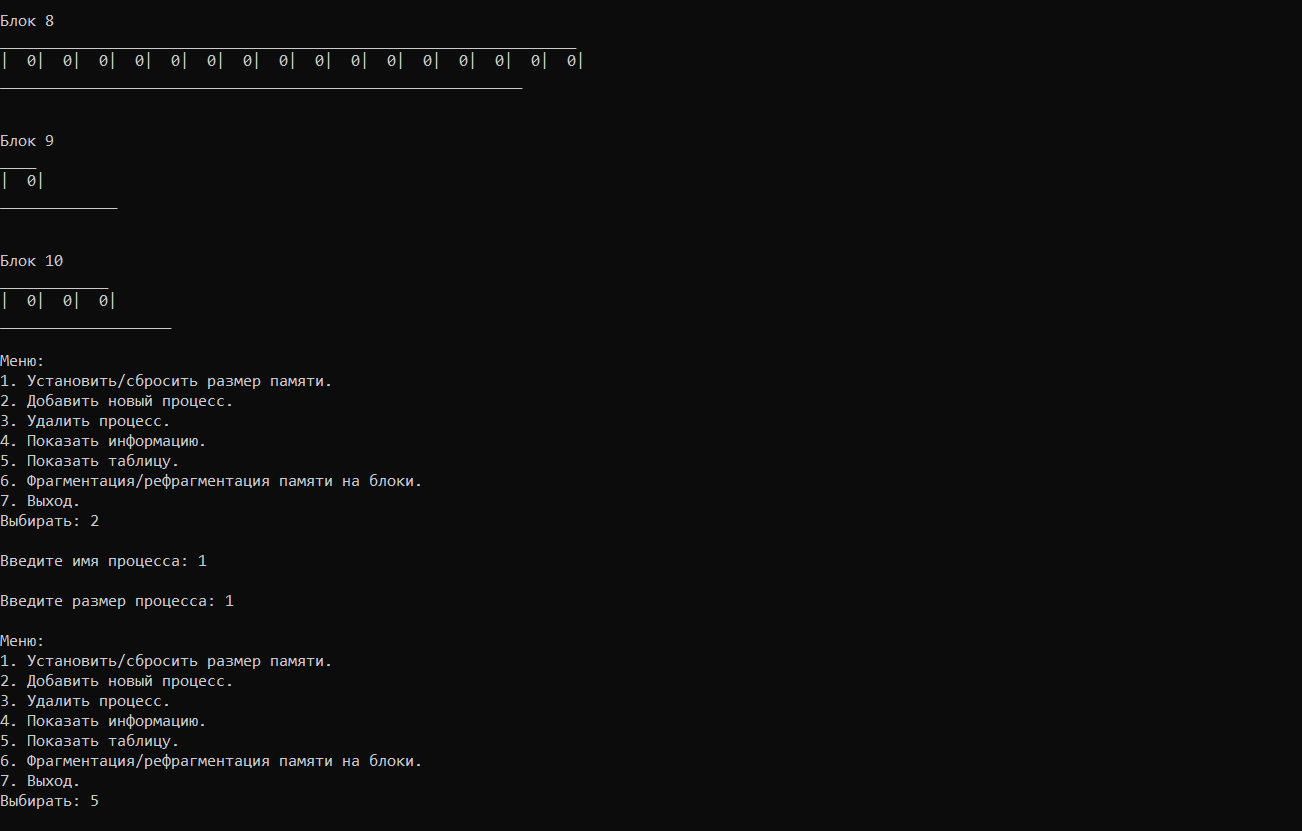
4. Реализовать возможность последовательной записи/чтения информации в/из выделенную память по логическому адресу. Вывести физического адреса ячейки памяти, в которую была осуществлена запись.

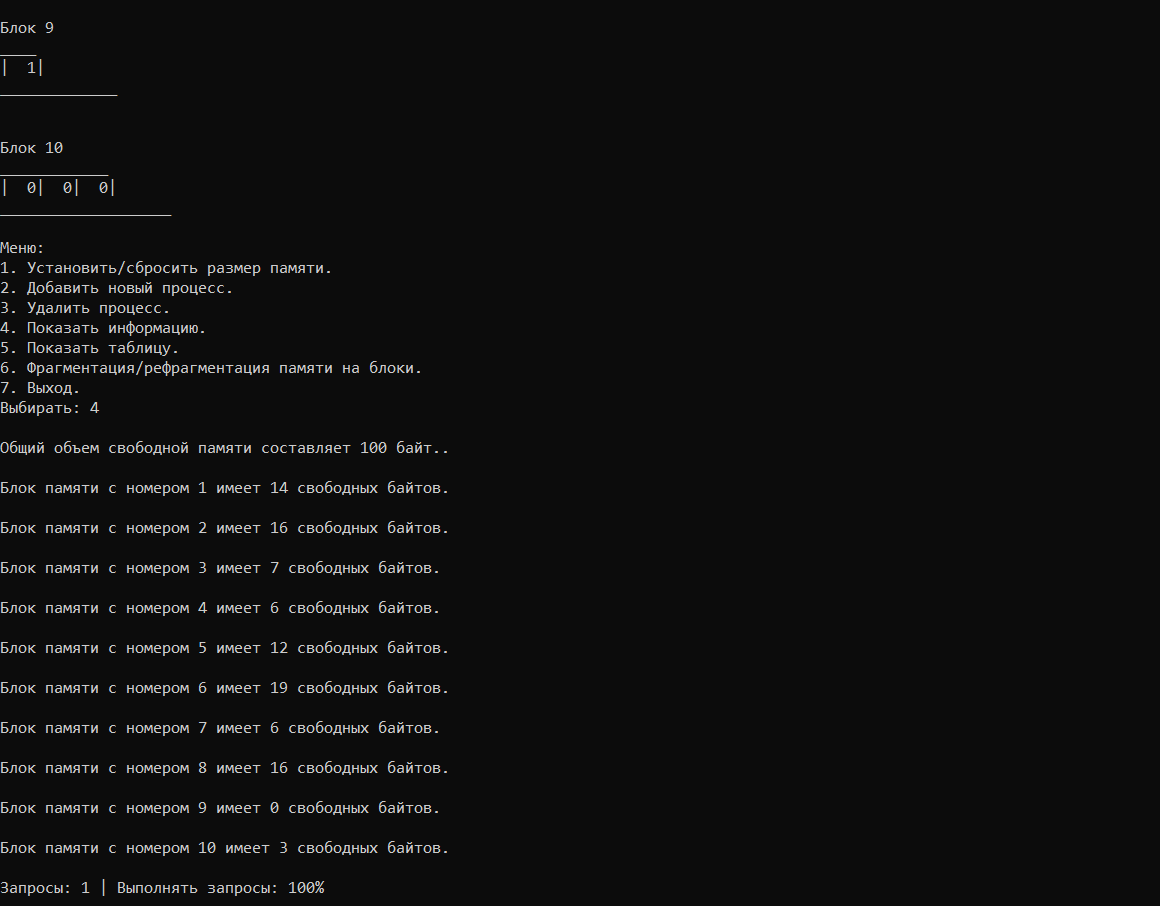
5. Организовывать циклическое выделение и освобождение памяти. При этом случайным образом задается количество выделяемых блоков и их размер.



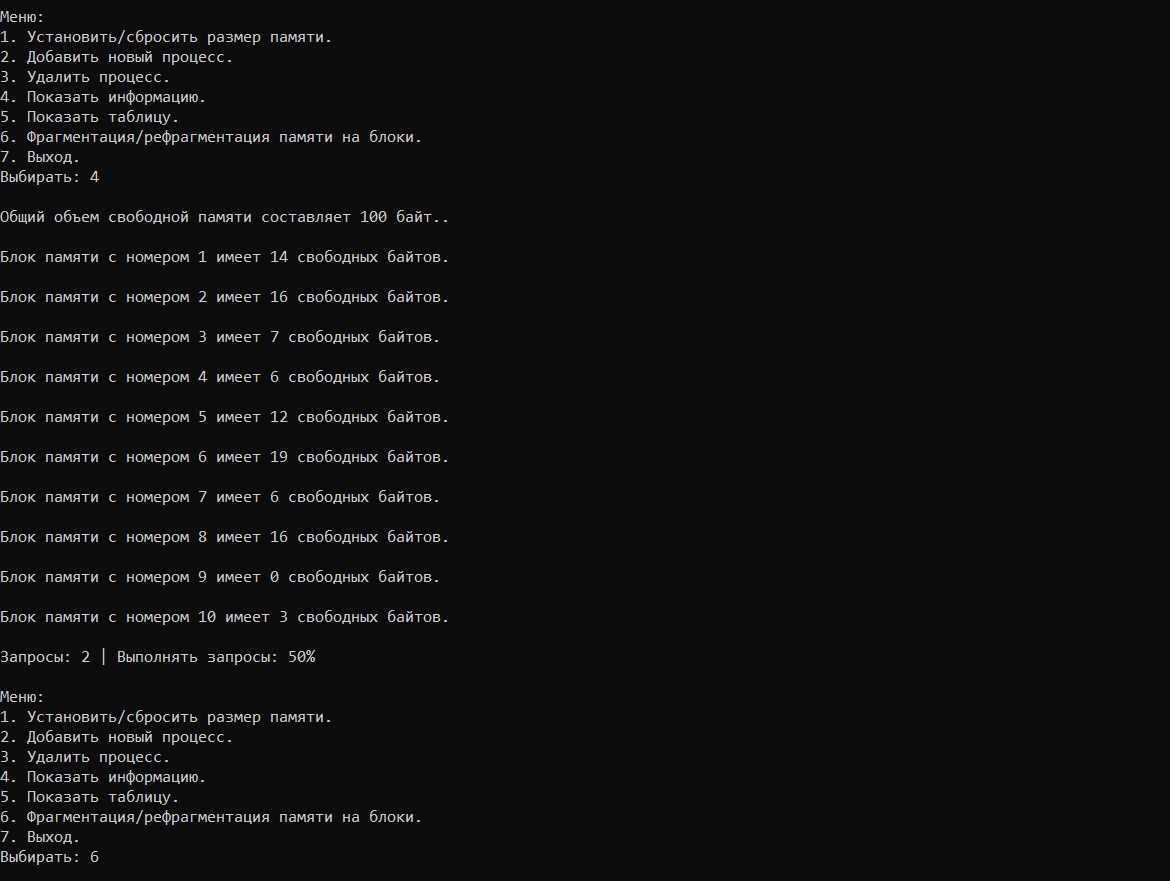
**Ход выполнения:**

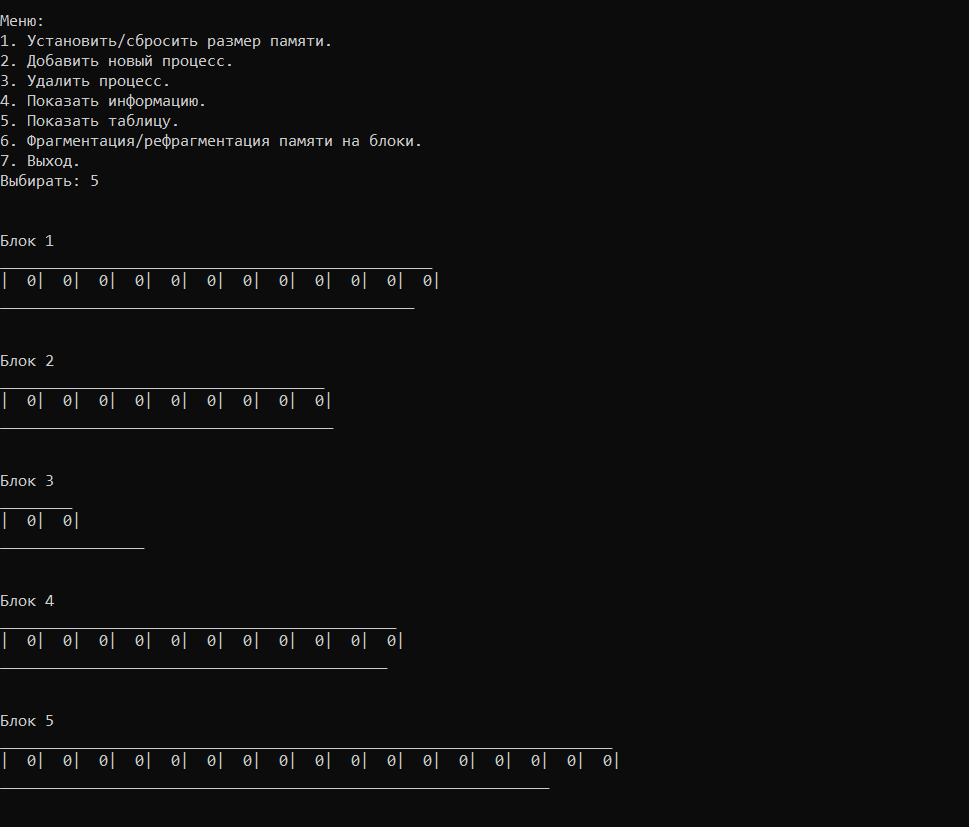
****

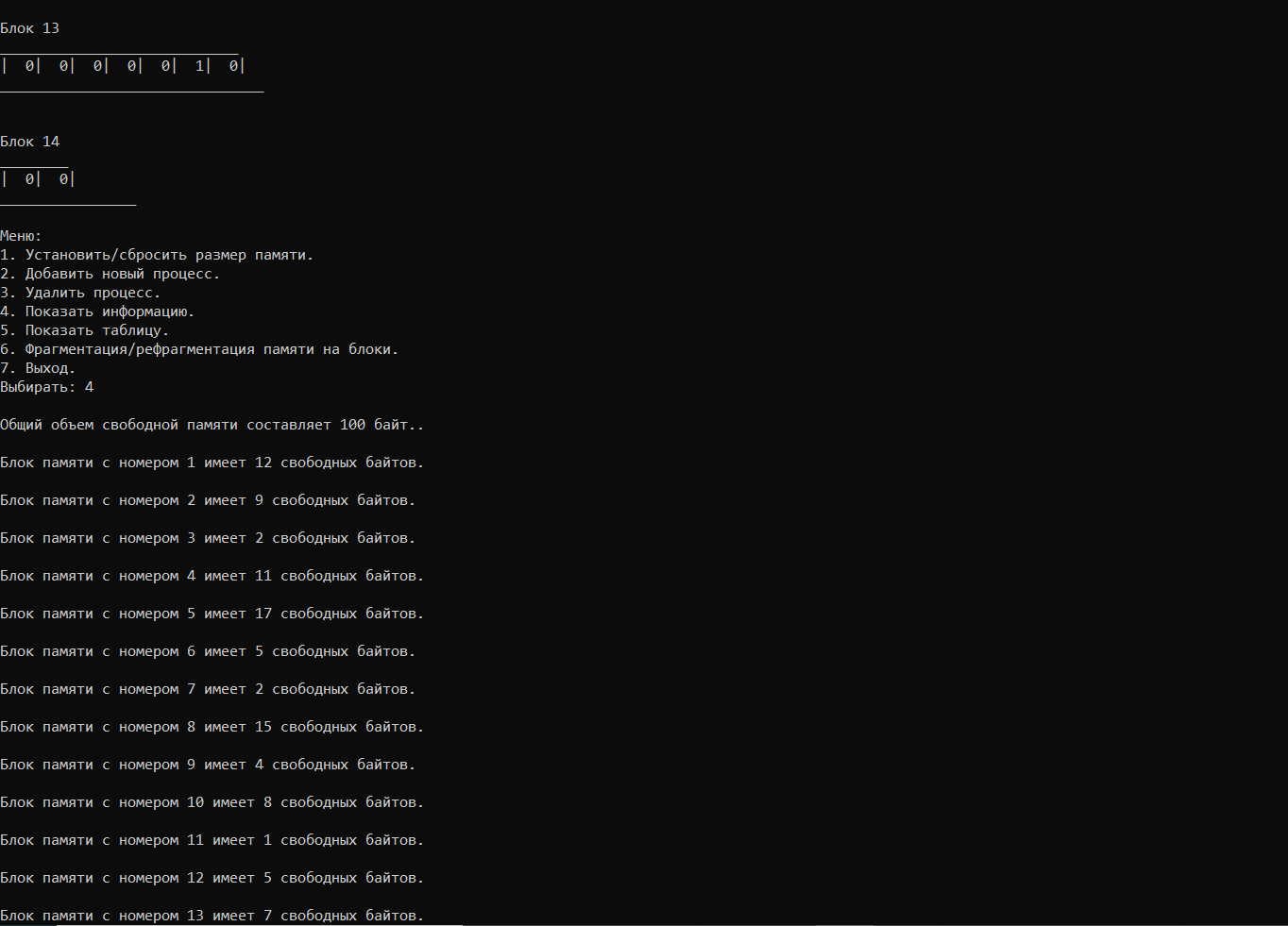
****

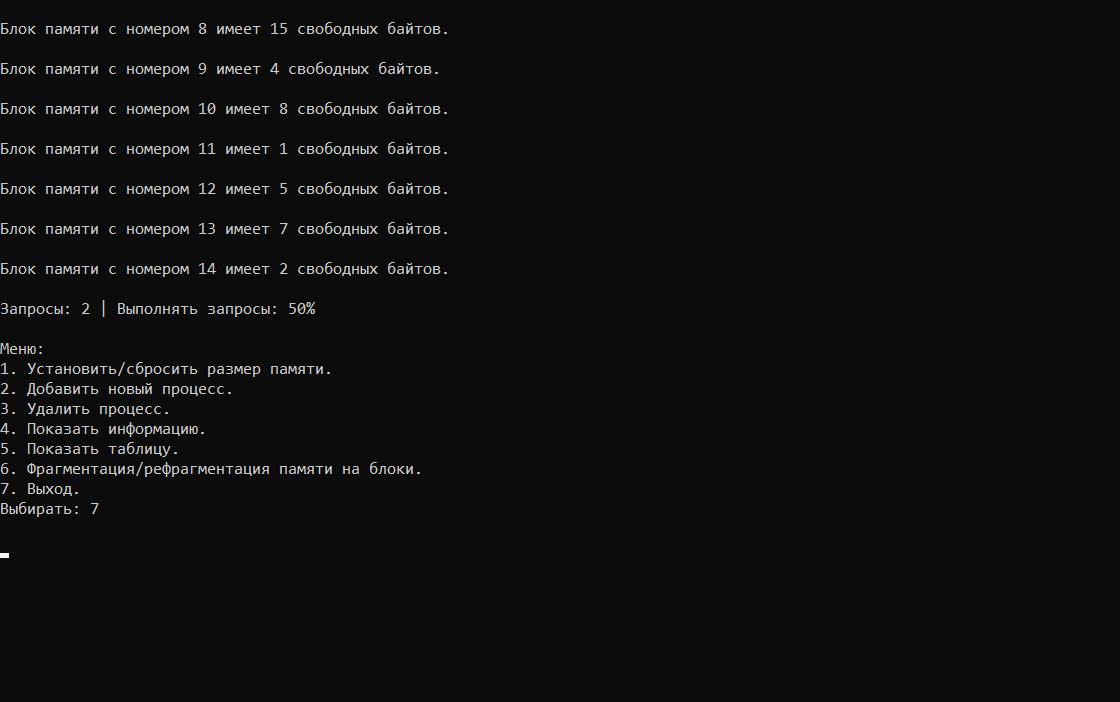
****

****

****

****

****

****

**Вывод работы:** в результате выполнения работы были изучены алгоритмы управления памятью, разработана программа менеджера памяти.

**Листинг программы:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <iostream>

typedef struct {

unsigned int name;

unsigned int size;

} process;

typedef struct {

unsigned int pointer;

unsigned int busy;

unsigned int size;

unsigned int size\_of\_process;

} block\_of\_memory;

unsigned int count = 0, count\_of\_blocks = 0;

process\* processes = NULL;

block\_of\_memory\* blocks = NULL;

unsigned int \*memory = NULL;

unsigned int request = 0, execute\_request = 0;

unsigned int size\_of\_memory = 0;

void table();

void fragmentation();

void allow\_mamory();

void add\_process();

void free\_process();

void info();

int main() {

setlocale(LC\_ALL, "rus");

unsigned int logic = 1;

while ( logic ) {

printf("Меню:\n");

printf("1. Установить/сбросить размер памяти.\n");

printf("2. Добавить новый процесс.\n");

printf("3. Удалить процесс.\n");

printf("4. Показать информацию.\n");

printf("5. Показать таблицу.\n");

printf("6. Фрагментация/рефрагментация памяти на блоки.\n");

printf("7. Выход.\n");

printf("Выбирать: ");

fflush(stdin);

switch( getchar() ) {

case '1' : {

free(memory);

printf("Введите размер памяти: ");

scanf("%d", &size\_of\_memory);

memory = (unsigned int \*)calloc( size\_of\_memory, sizeof( unsigned int ) ); // Initialization of Memory

break;

}

case '2' : {

add\_process();

break;

}

case '3' : {

free\_process();

break;

}

case '4' : {

info();

break;

}

case '5' : {

table();

break;

}

case '6' : {

fragmentation();

break;

}

case '7' : {

logic = 0;

break;

}

default : printf("\nВыберите пункт меню от 1 до 7!\n");

}

puts("");

}

// add\_process();

// info();

// table();

// free\_process();

// info();

// table();

// free\_process();

// info();

// table();

// for (unsigned int i = 0; i < count\_of\_blocks; i++) {

// printf("\n%d. Pointer: %d\n", i + 1, ( blocks + i ) -> pointer);

// printf("%d. Size: %d\n", i + 1, ( blocks + i ) -> size);

// printf("%d. Busy: %d\n", i + 1, ( blocks + i ) -> busy);

// }

free(memory);

if ( count != 0 ) free(processes);

free(blocks);

fflush(stdin);

getchar();

return(0);

}

void allow\_mamory() {

if ( count == 0 ) {

processes = (process\*)malloc( 1 \* sizeof( process ) );

} else {

processes = (process\*)realloc( processes, ( count + 1 ) \* sizeof( process ) );

}

count++;

}

void add\_process() {

setlocale(LC\_ALL, "rus");

request++;

unsigned int name\_test = 0, size\_test = 0, logic = 1, needed\_block = 0, size\_of\_needed\_block = size\_of\_memory + 1;

do {

logic = 0;

printf("\nВведите имя процесса: ");

scanf("%d", &name\_test );

for ( unsigned int i = 0; i < size\_of\_memory; i++ )

if ( \*( memory + i ) == name\_test ) {

printf("Процесс с таким именем существует! Введите другое имя процесса!\n");

logic = 1;

break;

}

} while ( logic );

puts("");

printf("Введите размер процесса: ");

scanf("%d", &size\_test );

for ( unsigned int i = 0; i < count\_of\_blocks; i++ )

if ( ( blocks + i ) -> busy == 0 && ( blocks + i ) -> size >= size\_test && ( blocks + i ) -> size - size\_test < size\_of\_needed\_block ) {

needed\_block = i;

size\_of\_needed\_block = ( blocks + i ) -> size - size\_test;

}

if ( size\_of\_needed\_block != size\_of\_memory + 1 ) {

( blocks + needed\_block ) -> busy = 1;

( blocks + needed\_block ) -> size\_of\_process = size\_test;

allow\_mamory();

( ( processes + count - 1 ) -> name ) = name\_test;

( ( processes + count - 1 ) -> size ) = size\_test;

for ( unsigned int j = ( blocks + needed\_block ) -> pointer; j < ( ( blocks + needed\_block ) -> pointer + ( blocks + needed\_block ) -> size\_of\_process ); j++ ) {

\*( memory + j ) = name\_test;

}

execute\_request++;

return;

}

printf("Процесс не может быть добавлен! Нет свободного места в памяти!\n");

}

void free\_process() {

setlocale(LC\_ALL, "rus");

unsigned int name\_of\_process = 0, pointer = 0;

printf("\nВведите имя процесса: ");

scanf("%d", &name\_of\_process);

for ( unsigned int i = 0; i < count; i++ )

if ( name\_of\_process == ( processes + i ) -> name ) {

for ( unsigned int j = i; i < count - 1; i++ ) {

( processes + i ) -> name = ( processes + i + 1 ) -> name;

( processes + i ) -> size = ( processes + i + 1 ) -> size;

}

if ( count > 1 ) processes = (process\*)realloc(processes, ( count - 1 ) \* sizeof( process ) );

else {

free(processes);

processes = NULL;

}

count--;

for ( unsigned int i = 0; i < count\_of\_blocks; i++ )

if ( \*( memory + ( blocks + i ) -> pointer ) == name\_of\_process ) {

for ( unsigned int j = ( blocks + i ) -> pointer; j < ( ( blocks + i ) -> pointer + ( blocks + i ) -> size\_of\_process ); j++ )

\*( memory + j ) = 0;

( blocks + i ) -> busy = 0;

( blocks + i ) -> size\_of\_process = 0;

}

return;

}

printf("Процесса с таким именем не существует!\n");

}

void info() {

setlocale(LC\_ALL, "rus");

unsigned int number = 1, count\_of\_free\_memory = 0;

printf("\nОбщий объем свободной памяти составляет %d байт..\n", size\_of\_memory);

for ( unsigned int i = 0; i < count\_of\_blocks; i++ )

printf("\nБлок памяти с номером %d имеет %d свободных байтов.\n", i + 1, ( blocks + i ) -> size - ( blocks + i ) -> size\_of\_process );

printf("\nЗапросы: %d | Выполнять запросы: %d%c\n", request, (unsigned int)( (float)execute\_request / request \* 100 ), '%' );

}

void fragmentation() {

unsigned int random\_size = 0, pointer\_on\_memory = 0, remake\_random = 0;

for ( unsigned int i = time(NULL); i < time(NULL) + 100; i++ ) {

remake\_random += rand() % rand();

}

srand( remake\_random );

count\_of\_blocks = 0;

free(blocks);

while ( pointer\_on\_memory < size\_of\_memory ) {

random\_size = rand() % 19 + 1;

if ( random\_size + pointer\_on\_memory <= size\_of\_memory ) {

if ( count\_of\_blocks == 0 ) blocks = (block\_of\_memory\*)malloc( 1 \* sizeof( block\_of\_memory ) );

else blocks = (block\_of\_memory\*)realloc( blocks, ( count\_of\_blocks + 1 ) \* sizeof( block\_of\_memory ) );

( blocks + count\_of\_blocks ) -> pointer = pointer\_on\_memory;

( blocks + count\_of\_blocks ) -> size = random\_size;

( blocks + count\_of\_blocks ) -> busy = 0;

( blocks + count\_of\_blocks ) -> size\_of\_process = 0;

count\_of\_blocks++;

pointer\_on\_memory += random\_size;

}

}

}

void table() {

for ( unsigned int i = 0; i < count\_of\_blocks; i++ ) {

printf("\n\n");

printf("Блок %d\n", i + 1);

printf("\xDA\xC4");

for ( unsigned int j = 0; j < ( blocks + i ) -> size - 1; j++ ) printf("\xC4\xC4\xC2\xC4");

printf("\xC4\xC4\xBF\n");

for ( unsigned int j = 0; j < ( blocks + i ) -> size; j++ ) printf("\xB3%3d", \*( memory + ( blocks + i ) -> pointer + j ) );

printf("\xB3\n");

printf("\xC0\xC4");

for ( unsigned int j = 0; j < ( blocks + i ) -> size - 1; j++ ) {

printf("\xC4\xC4\xC1\xC4");

}

printf("\xC4\xC4\xD9\n");

}

}