УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8

по дисциплине **«Операционные системы»**

на тему: «Простейшие схемы управления памятью»

Выполнил: студент гр. ИТ-11
Иванов А.А.
Принял: преподаватель
Петров И.И.

Гомель 2021

Цель работы: изучение алгоритмов управления памятью, разработка программы менеджера памяти.

Теоретические сведения

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Схема с переменными разделами

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 (рис. 8.6). Смежные свободные участки могут быть объединены.

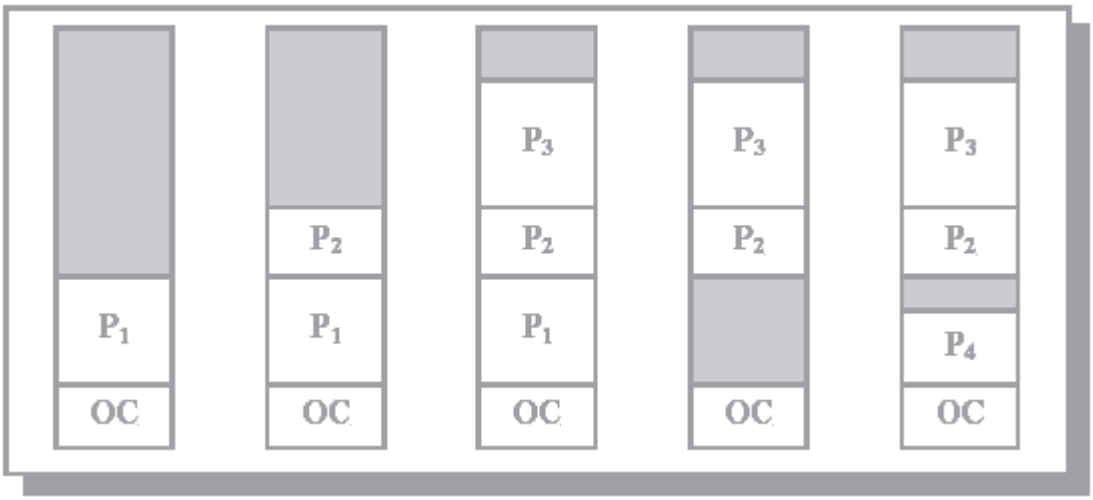


Рис. 8.6. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

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.

Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 8.7. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

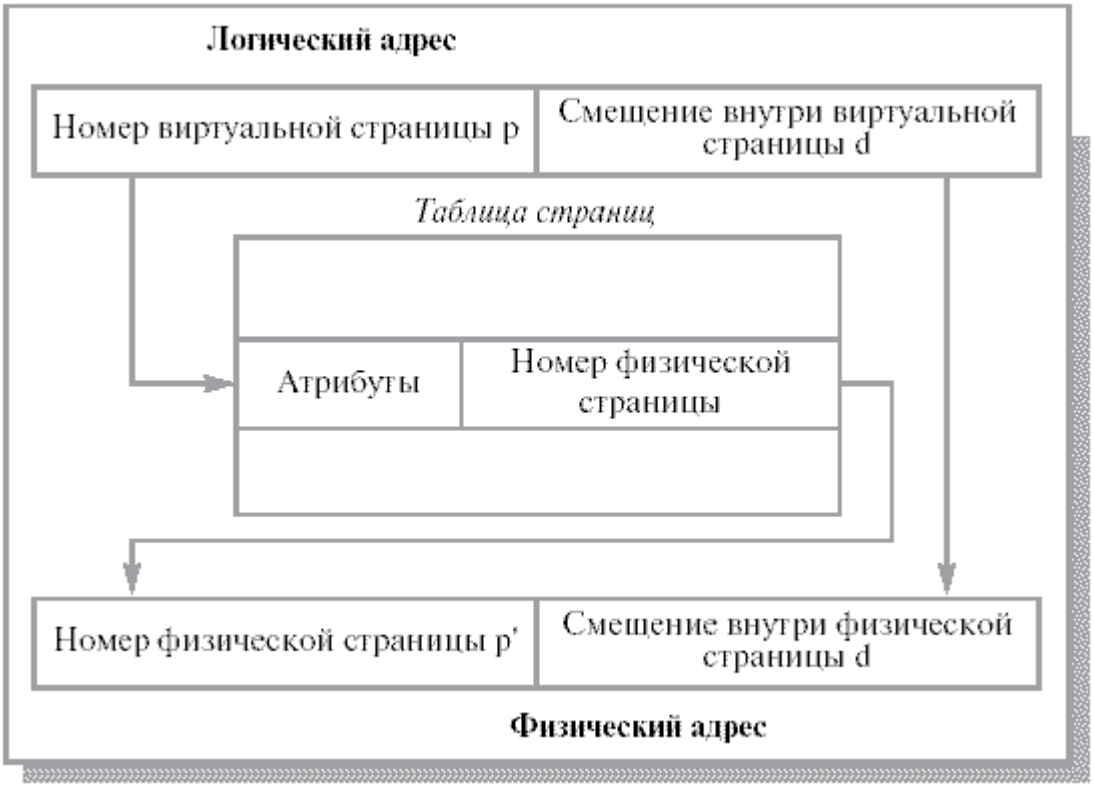


Рис. 8.7. Связь логического и физического адресов при страничной организации памяти

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое

непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, **что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...)**.

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 2^{32} байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе

таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длинасегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

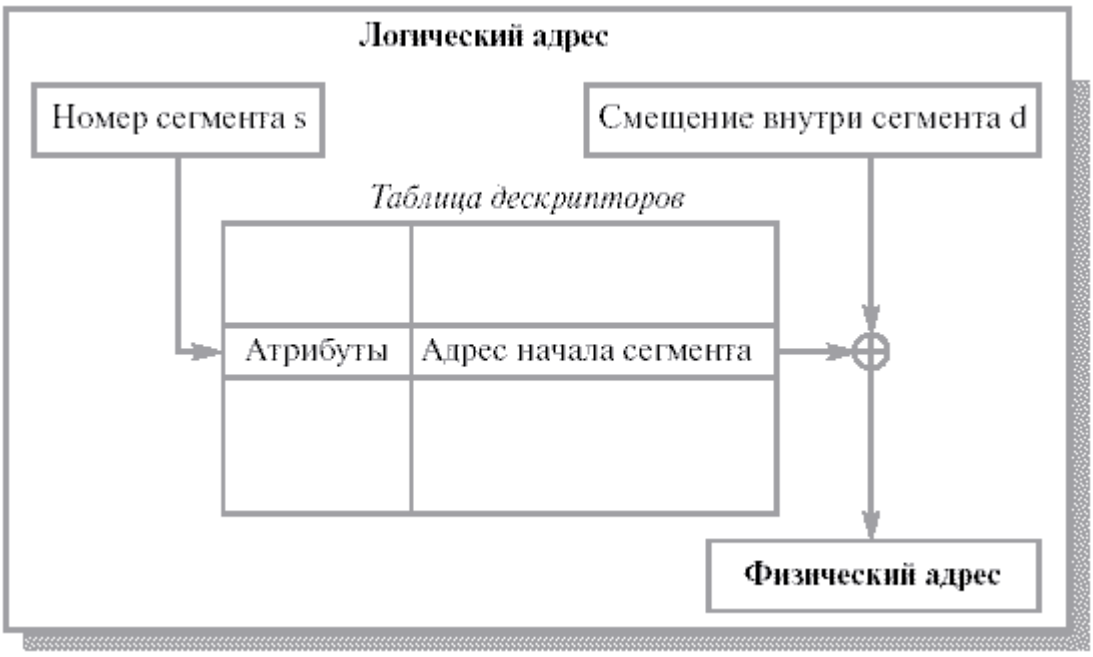


Рис. 8.8. Преобразование логического адреса при сегментной организации памяти

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей:

номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

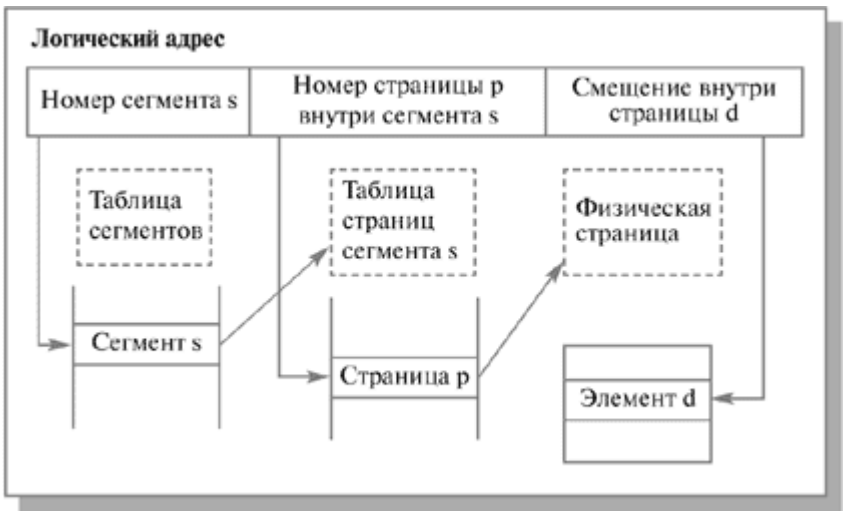


Рис. 8.9. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и сегментная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

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, объём свободной памяти, размер наибольшего свободного блока, количество запросов на выделение памяти, количество удовлетворённых запросов (%).
2. Для выделения памяти указывается имя процесса и размер блока. После нажатия на кнопку «ДОБАВИТЬ» память выделяется или выдаётся 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.

3	**Свопинг.** Выгружается процесс, занимающий наименьший объём памяти.

Листинг программы

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class D
    {
        public static int V, V_sv, n, k_a = 0, k_success = 0;
        public static Process[] proc;
        public static Process[] sw = new Process[20];
        public static void Create()
        {
            int t;
           Console.Write("\nВведите количество разделов памяти: ");
            n = Convert.ToInt32(Console.ReadLine());
            proc = new Process[n];
            Console.Write("Введите объём раздела памяти: ");
            t = Convert.ToInt32(Console.ReadLine());
            V = n * t;
            for (int i = 0; i < n; i++)
                proc[i] = new Process(t);
            V_sv = V;
            for (int i = 0; i < sw.Length; i++)
                sw[i] = new Process(t);
        }
        public static int Swap()
        {
            Console.Clear();
            int min = V;
            int r = -1;
            for (int i = 0; i < proc.Length; i++)
            {
                if (proc[i].Mem_p != 0 && proc[i].Mem_p < min) { min = proc[i].Mem_p; r = i; }
            }
            if (r != -1)
            {
                int g = 0, ff = 0;
                while (g < sw.Length && ff == 0)
                {
                    if (sw[g].Name_p == null)
                    {
                        sw[g].Name_p = proc[r].Name_p;
                        sw[g].Mem_p = proc[r].Mem_p;
                        sw[g].data = proc[r].data;
                        ff = 1;
                    }
                    g++;
                }
                proc[r].Mem_p = 0;
                proc[r].Name_p = null;
                proc[r].data = null;
                V_sv += proc[r].V0;
                Console.WriteLine("Процесс, занимающий наименьший объём памяти выгружен");
            }
            else { Console.WriteLine("Процессов не обнаружено"); Console.ReadKey(); }
            return r;
        }
        public static void add()
        {
```

```
            Console.Clear();
            string name;
            int vv;
            Console.Write("\nВведите имя процесса: ");
            name = Console.ReadLine();
            Console.Write("\nВведите объём памяти требуемой данному процессу(Кб): ");
            vv = Int32.Parse(Console.ReadLine());
            k_a++;
            if (vv > V) Console.WriteLine("Ошибка: объём всей памяти меньше объёма памяти
требуемой процессу");
            else
            {
                if (vv <= proc[0].V0)
                {
                    int f = 0, i = 0;
                    while (i < proc.Length && f == 0)
                    {
                        if (proc[i].Name_p == null)
                        {

                            proc[i].Name_p = name;
                            proc[i].Mem_p = vv;
                            k_success++; f++;
                            proc[i].data = new string[vv];
                            V_sv = V_sv - proc[i].V0;
                        }

                        i++;
                    }

                    if (f == 0)
                    {
                        int r = Swap();
                        proc[r].Name_p = name;
                        proc[r].Mem_p = vv;
                        k_success++;
                        proc[r].data = new string[vv];
                        V_sv = V_sv - proc[r].V0;

                    }
                }
                else { Console.WriteLine("Ошибка: объём раздела меньше объёма требуемой памяти");
Console.ReadKey(); }
            }

        }
        public static void Vyvod()
        {
            Console.WriteLine("\nИнформация о процессах\n");
            int f = 0;
            for (int i = 0; i < proc.Length; i++)
            {

                if (proc[i].Mem_p != 0)
                { Console.WriteLine("Процесс: {0,-9}Объём памяти(Кб): {1,-5}  Номер блока памяти:
{2} ", proc[i].Name_p, proc[i].Mem_p, i + 1); f++; }

            }
            if (f == 0) Console.WriteLine("Нет процессов.");

        }
        public static void SW()
        {
            Console.WriteLine("\nВыгруженные процессы\n");
            int f = 0;
            for (int i = 0; i < sw.Length; i++)
            {
                if (sw[i].Mem_p != 0)
                { Console.WriteLine("Процесс: {0,-9}Объём памяти(Кб): {1,-5} ", sw[i].Name_p,
sw[i].Mem_p); f++; }
            }
            if (f == 0) Console.WriteLine("Нет процессов.");

        }
        public static void MaxR()
        {
            int temp;
            int max = -1;
            for (int i = 0; i < proc.Length; i++)
            {
```

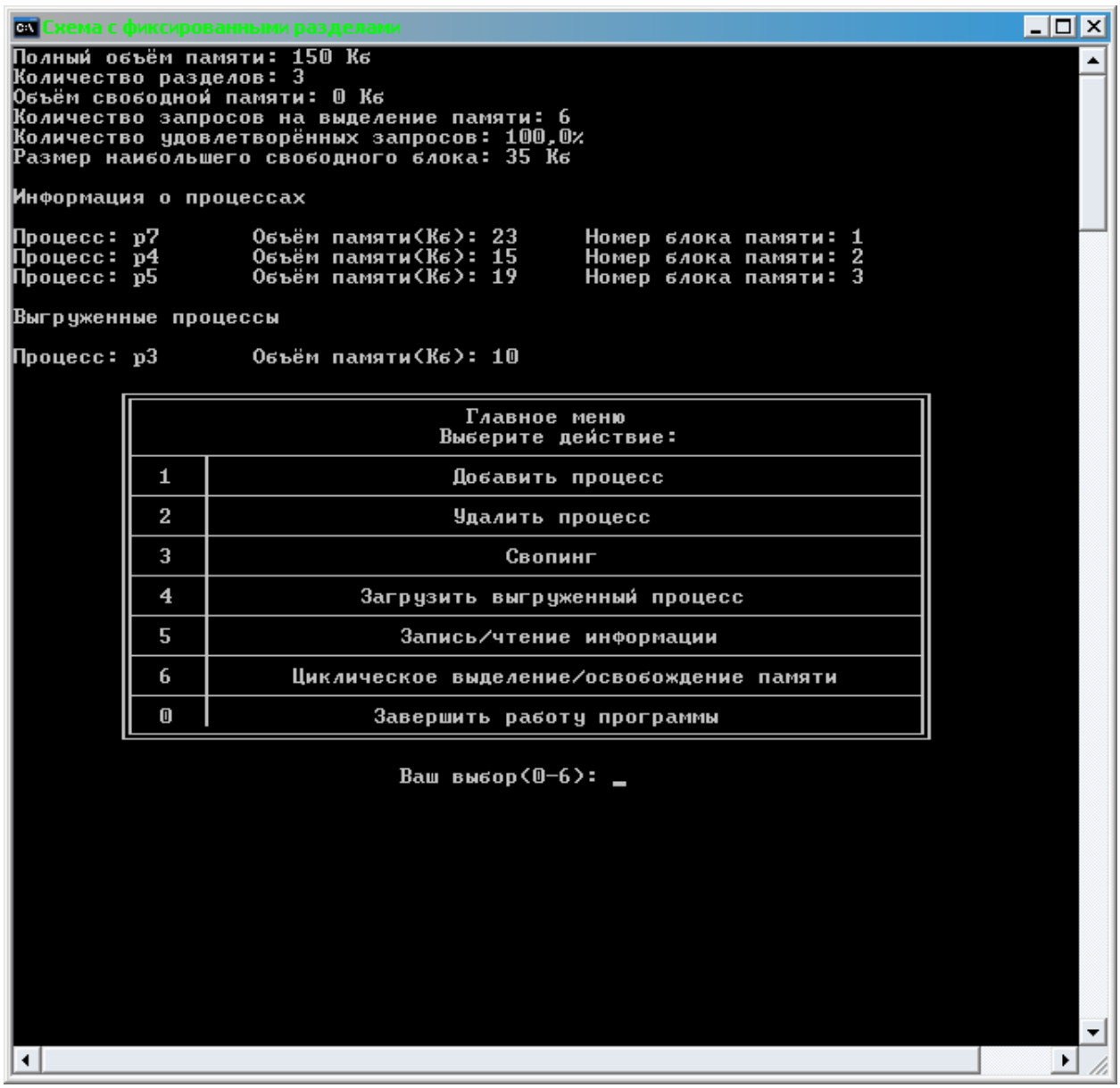
```
                temp = proc[i].V0 - proc[i].Mem_p;
                if (temp > max) max = temp;
            }
            if (max != -1) Console.WriteLine("Размер наибольшего свободного блока: " + max + "
Кб");
        }
        public static void KolPercent()
        {
            if (k_a != 0)
            {
                float k = 100 * (float)k_success / k_a;
                Console.WriteLine("Количество удовлетворённых запросов: {0,-4:f1}%", k);
            }
        }
        public static void Remove()
        {
            Console.Clear();
            if (V_sv != V)
            {
                Vyvod();
                int f = 0;
                int nom;
                string name;
                Console.Write("\nВведите имя процесса: ");
                name = Console.ReadLine();
                Console.Write("Введите номер блока: ");
                nom = Convert.ToInt32(Console.ReadLine());
                nom--;
                int i = 0;
                while (i < proc.Length && f == 0)
                {
                    if (proc[i].Name_p == name && i == nom)
                    {
                        proc[i].Name_p = null; proc[i].Mem_p = 0; proc[i].data = null;
                        f = 1; V_sv += proc[i].V0;
                        Console.WriteLine("Процесс удалён");
                    }
                    i++;
                }
                if (f == 0) Console.WriteLine("Процесс не найден");
            }
            else Console.WriteLine("Процессов не обнаружено");
            Console.ReadKey();

        }
        public static void ZapUd()
        {
            Console.Clear();
            if (V_sv != V)
            {
                Vyvod();
                int t = 0;
                int f = 0;
                int nom;
                string name;
                Console.WriteLine("\nЗапись\\чтение информации в выделенную память\n");
                Console.WriteLine("1.Запись\n2.Чтение");
                Console.Write("Ваш выбор: ");
                t = Convert.ToInt32(Console.ReadLine());
                switch (t)
                {
                    case 1:
                        {
                            int j = 0;
                            Console.Write("\n\nВведите имя процесса: ");
                            name = Console.ReadLine();
                            Console.Write("Введите номер блока: ");
                            nom = Convert.ToInt32(Console.ReadLine());
                            nom--;
                            for (int i = 0; i < proc.Length; i++)
                                if (proc[i].Name_p == name && i == nom)
                                {
                                    Console.WriteLine("\nЗапись информации");

                                    while (j < proc[i].data.Length && f == 0)
                                    {
                                        if (proc[i].data[j] == null)
                                        {
                                            Console.WriteLine("по логическому адресу: " + j);
f++;
```

```
                                        proc[i].data[j] = Console.ReadLine();

                                        int k = 0, S = 0;
                                        while (k < i)
                                        {
                                            S += proc[k].data.Length;
                                            k++;
                                        }
                                        Console.WriteLine("Физический адрес: " + (S + j));

                                    }
                                    j++;
                                }

                            }
                            if (j == 0) Console.WriteLine("Процесс не найден");
                        }
                        break;
                    case 2:
                        {

                            Console.Write("\n\nВведите имя процесса: ");
                            name = Console.ReadLine();
                            Console.Write("Введите номер блока: ");
                            nom = Convert.ToInt32(Console.ReadLine());
                            nom--;
                            int i = 0, p = 0;
                            while (i < proc.Length && p == 0)
                            {
                                if (proc[i].Name_p == name && i == nom)
                                {
                                    Console.WriteLine("\nВывод информации, хранящейся в
выделенной для данного процесса памяти");
                                    for (int ii = 0; ii < proc[i].data.Length; ii++)
                                        Console.WriteLine("Логический адрес: " + ii + " Данные: "
+ proc[i].data[ii]);
                                    p++;
                                }
                                i++;
                            }

                        }
                        break;
                    default: break;
                }

            }
            else Console.WriteLine("Процессов не обнаружено");
            Console.ReadKey();
        }
        public static void VidOsv()
        {

            int tt = 0, nom;
            int rand = 0;

            Random n1 = new Random();
            Random nn = new Random();
            bool b = true;
            int ppp = n1.Next(1, 7); // кол-ва разделов
            proc = new Process[ppp];
            int ppp11 = n1.Next(10, 50); // Размер блока

            V = ppp * ppp11;
            for (int i = 0; i < ppp; i++)
                proc[i] = new Process(ppp11);
            n = proc.Length;
            V_sv = V;
            k_a = 0; k_success = 0;
            while (b)
            {
                Console.Clear();

                rand = nn.Next(1, proc[0].V0);
                if (rand % 2 == 0) // если чётное, то память выделяется
                {

                    #region
                    int f = 0, i = 0;
                    while (i < proc.Length && f == 0)
```

```
            {
                k_a++;
                if (proc[i].Name_p == null)
                {

                    proc[i].Name_p = n1.Next(1, 1000).ToString();
                    proc[i].Mem_p = rand;
                    k_success++; f++;
                    proc[i].data = new string[rand];
                    V_sv = V_sv - proc[i].V0;
                }

                i++;
            }

            #endregion
        }
        else
        {
            #region
            if (V_sv != V)
            {
                int f = 0;
                nom = n1.Next(0, proc.Length - 1);
                int i = 0;
                while (i < proc.Length && f == 0)
                {
                    if (proc[i].Name_p != null && i == nom)
                    {
                        proc[i].Name_p = null; proc[i].Mem_p = 0; proc[i].data = null;
                        f = 1; V_sv += proc[i].V0;
                    }
                    i++;
                }
            }
            else Console.WriteLine("Процессов не обнаружено");
            #endregion
        }

        Vyvod();

        Console.WriteLine("\n\nОстановить?");
        Console.WriteLine("1.Да\n2.Нет");
        Console.Write("Ваш выбор: ");
        tt = Convert.ToInt32(Console.ReadLine());
        switch (tt)
        {
            case 1: b = false; break;
            case 2: b = true; break;
            default: break;
        }
    }

}
public static void Load()
{
    int i = 0, f = 0;
    if (sw[0].Name_p != null)
    {
        while (i < proc.Length && f == 0)
        {
            if (proc[i].Name_p == null)
            {
                proc[i].Name_p = sw[0].Name_p;
                proc[i].Mem_p = sw[0].Mem_p;
                proc[i].data = sw[0].data;
                V_sv = V_sv - proc[i].V0;
                f = 1;
                for (int k = 0; k < sw.Length - 1; k++)
                {
                    sw[k].Name_p = sw[k + 1].Name_p;
                    sw[k].Mem_p = sw[k + 1].Mem_p;
                    sw[k].data = sw[k + 1].data;
                }
            }

            i++;
        }
        if (f == 0)
        {
```

```
                Console.WriteLine("Все разделы заняты Выполнить свопинг?");
                Console.WriteLine("1.Да\n2.Нет");
                Console.Write("Ваш выбор: ");
                int t = 0;
                t = Convert.ToInt32(Console.ReadLine());
                switch (t)
                {
                    case 1:
                        {
                            int r;


                            r = Swap();

                            proc[r].Name_p = sw[0].Name_p;
                            proc[r].Mem_p = sw[0].Mem_p;
                            proc[r].data = sw[0].data;
                            V_sv = V_sv - proc[r].V0;

                            for (int k = 0; k < sw.Length - 1; k++)
                            {

                                sw[k].Name_p = sw[k + 1].Name_p;
                                sw[k].Mem_p = sw[k + 1].Mem_p;
                                sw[k].data = sw[k + 1].data;
                            }

                        }
                        break;
                    default: break;
                }

            }

        }
        else { Console.WriteLine("Выгруженных процессов нет"); Console.ReadKey(); }
    }
}
class Process
{
    int mem_p, V_razd;
    string name_p;
    public Process(int V_r) { V_razd = V_r; }
    public string[] data;
    public int V0
    {
        get { return V_razd; }
        set { V_razd = value; }
    }
    public int Mem_p
    {
        get { return mem_p; }
        set { mem_p = value; }
    }
    public string Name_p
    {
        get { return name_p; }
        set { name_p = value; }
    }

}

class Program
{
    static void Main(string[] args)
    {
        Console.Title = "8 ";
        int k = 12;
        D.Create();
        do
        {
            Console.Clear();
            Console.WriteLine("Полный объём памяти: " + D.V + " Кб");
            Console.WriteLine("Количество разделов: " + D.n);
            Console.WriteLine("Объём свободной памяти: " + D.V_sv + " Кб");
            Console.WriteLine("Количество запросов на выделение памяти: " + D.k_a);
            D.KolPercent();
            D.MaxR();
            D.Vyvod();
            D.SW();
```

```
Console.WriteLine("\n\t╔═══════════════════════════════════════════════════════════╗");
                Console.WriteLine("\t║                          Главное меню
║");
                Console.WriteLine("\t║                       Выберите действие:
║");

Console.WriteLine("\t╠═══╤═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  1  │                  Добавить процесс
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  2  │                  Удалить процесс
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  3  │                      Свопинг
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  4  │          Загрузить выгруженный процесс
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  5  │             Запись/чтение информации
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  6  │      Циклическое выделение/освобождение памяти
║");

Console.WriteLine("\t╠═══╪═══════════════════════════════════════════════════════╣");
                Console.WriteLine("\t║  0  │           Завершить работу программы
║");

Console.WriteLine("\t╚═══╧═══════════════════════════════════════════════════════╝");
                Console.Write("\n\t\t\t     Ваш выбор(0-6): ");
                k = Convert.ToInt32(Console.ReadLine());
                switch (k)
                {
                    case 1: D.add(); break;
                    case 2: {D.Remove(); D.Load();} break;
                    case 3: D.Swap(); break;
                    case 4: D.Load(); break;
                    case 5: D.ZapUd(); break;
                    case 6: D.VidOsv(); break;
                    case 0: Console.Clear(); Console.WriteLine("Выход..."); break;
                    default: break;
                }
            }
            while (k != 0);
            Console.ReadKey();
        }
    }
}
```



Вывод: изучение алгоритмов управления памятью, разработка программы менеджера памяти.