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: «Управление виртуальной памятью»

Выполнил: студент гр. ИТ-11

Иванов А.А.

Принял: преподаватель

Петров И.И.

Гомель 2021

Цель работы: ознакомится с виртуальной памятью, её свойствами и возможностями.

Теоретические сведения

Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на *компьютер*. Одним из главных достижений стало появление *виртуальной памяти* (*virtual memory*). Впервые она была реализована в 1959 г. на компьютере "Атлас", разработанном в Манчестерском университете.

Суть концепции ***виртуальной памяти*** заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах *виртуальной памяти* у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И, наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный *бит присутствия*, входящий в состав атрибутов страницы в *таблице страниц* .

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. *Принцип локальности* обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

- Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
- Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
- Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы "видимости" практически неограниченной (характерный размер для 32-разрядных архитектур 2^{32} = 4 Гбайт) адресуемой пользовательской памяти (логическое *адресное пространство*) при наличии основной памяти существенно меньших размеров (физическое *адресное пространство*) – очень важный аспект.

Но введение *виртуальной памяти* позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими *виртуальными адресами*, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, *пользовательский процесс* лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный *компьютер* PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. *Операционная система* этого компьютера тем не менее поддерживала *виртуальную память*, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с *виртуальной памятью* те адреса, которые генерирует *программа* (логические адреса), называются виртуальными, и они формируют *виртуальное адресное пространство*. Термин " ***виртуальная память*** " означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации *виртуальной памяти*, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае *виртуальной памяти* это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (*page fault*).

Любая из трех ранее рассмотренных схем управления памятью – *страничной*, сегментной и *сегментно-страничной* – пригодна для организации *виртуальной памяти*. Чаще всего используется сегментно-*страничная модель*, которая является синтезом *страничной модели* и идеи сегментации. Причем для тех архитектур, в которых *сегменты* не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый *компонент* управления *виртуальной памятью*. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением *виртуальной памятью*, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления *виртуальной памятью* на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления *виртуальной памятью*. Компоненты аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое *виртуальное адресное пространство* в физическое *адресное пространство* существенно меньшего размера. *Пользовательский процесс* или ОС должны иметь возможность

осуществить *запись* по *виртуальному адресу*, а задача ОС – сделать так, чтобы записанная *информация* оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю *память*). В случае *виртуальной памяти* система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области *виртуальной памяти* в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память

Как и в случае простой ***страничной организации***, страничная *виртуальная память* и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. *Виртуальные адреса* делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной *виртуальной памяти* называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы *виртуальный адрес* преобразуется в упорядоченную пару (p,d), где p – номер страницы в *виртуальной памяти*, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, *таблица страниц* принимает специфический вид (см. раздел "Структура *таблицы страниц* "), структура записей становится более сложной, среди атрибутов страницы появляются *биты присутствия*, *модификации* и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая *страничное нарушение* (page fault) или страничный отказ. Обработка *страничного нарушения* заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных

кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

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

В большинстве современных компьютеров со *страничной организацией* в основной памяти хранится лишь часть *таблицы страниц*, а быстрота доступа к элементам таблицы текущей *виртуальной памяти* достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах *виртуальной памяти* сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации *виртуальной памяти* размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей *таблицы страниц* для каждого сегмента.

На практике, однако, появления в системе большого количества *таблиц страниц* стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной *таблицы страниц*. Таким образом, одна *таблица страниц*отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным *виртуальным адресам* вне зависимости от выполняемого процесса.

Структура таблицы страниц

Организация *таблицы страниц* – один из ключевых элементов отображения адресов в *страничной* и *сегментно-страничной* схемах. Рассмотрим структуру *таблицы страниц* для случая *страничной организации* более подробно.

Итак, *виртуальный адрес* состоит из виртуального номера страницы и смещения. Номер записи в *таблице страниц* соответствует номеру *виртуальной страницы*. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в *таблице страниц* находится номер кадра для данной *виртуальной страницы*, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в *таблице страниц* содержит информацию об атрибутах страницы. Это *биты присутствия* и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: ***бит модификации***, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; ***бит ссылки***, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью *таблицы страниц*.

Основную проблему для эффективной реализации *таблицы страниц* создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 2^{64} байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой *ассоциативной памяти* (см. следующий раздел).

Подсчитаем примерный размер *таблицы страниц*. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем $2^{32}/2^{12}=2^{20}$, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей *таблице страниц* (а в случае *сегментно-страничной* схемы желательно иметь по одной *таблице страниц* на каждый сегмент).

Понятно, что количество памяти, отводимое *таблицам страниц*, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты *таблицы страниц*. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение *таблицы*

страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой *многоуровневой таблицы страниц*. Для примера рассмотрим двухуровневую *таблицу* с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} *таблиц* второго уровня по 2^{10} строк. Эти *таблицы* второго уровня объединены в общую структуру при помощи одной *таблицы* первого уровня, состоящей из 2^{10} строк. 32-разрядный адрес делится на 10-разрядное поле p_1, 10-разрядное поле p_2 и 12-разрядное смещение d. Поле p_1 указывает на нужную строку в *таблице* первого уровня, поле p_2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. рис. 9.1).

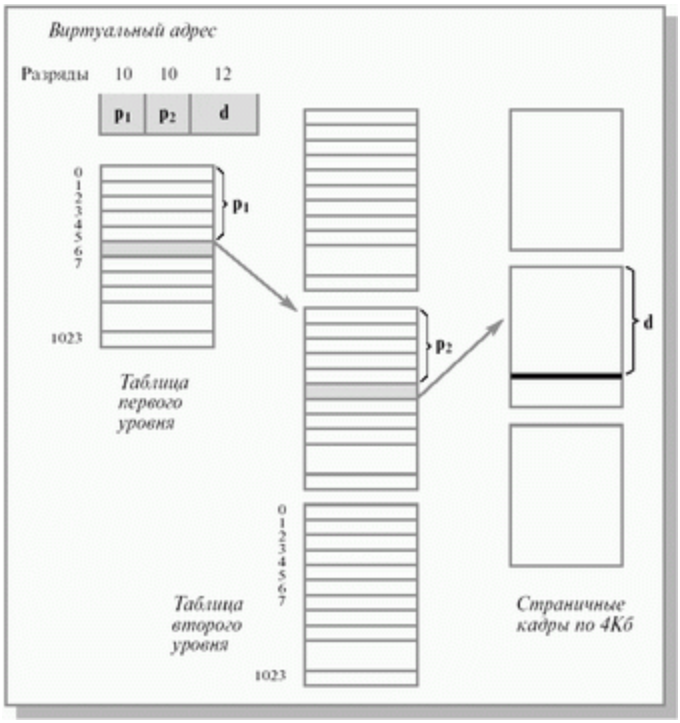


Рис. 9.1. Пример двухуровневой таблицы страниц

При помощи всего лишь одной *таблицы* второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну *таблицу* первого уровня и несколько *таблиц* второго уровня. Очевидно, что суммарное количество строк в этих *таблицах* много меньше 2^{20}. Такой подход естественным образом обобщается на три и более уровней *таблицы*.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры *таблиц* на каждом уровне подобраны так, чтобы *таблица* помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в *таблице страниц* зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) *пейджинга*, а также *пейджинга* с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

Задание: разработать программу, реализующую заданный алгоритм замещения страниц в памяти.

Задание
Разработать программу, реализующую заданный алгоритм замещения страниц в памяти.
Менеджер памяти должен:
1. Разбивать память заданного размера на указанное количество страниц. На экран должна выводиться следующая информация о состоянии памяти: объем памяти, число страниц, число свободных страниц (%), размер страницы;
2. Размещать в памяти страницу заданного процесса, с замещением занятой по заданному алгоритму (по нажатию кнопки «ДОБАВИТЬ»). Для размещения страницы в памяти, указывается имя процесса и ее номер (вводятся отдельно).
Например: Pro 3. После нажатия на кнопку «ДОБАВИТЬ» страница размещается в свободной странице памяти. Если задано **глобальное размещение** (см. вариант задания), то выбирается любая не занятая страница. При **локальном размещении** страница размещается только среди виртуальных страниц выделенных этому процессу. Выделение страниц в памяти выполняется при первом ее занесении процесса в память. Алгоритм

замещения выполняется **только при отсутствии свободных страниц** под процесс.;
3. Удалять из памяти заданную страницу или все страницы заданного процесса (по нажатию кнопки «УДАЛИТЬ»). Указывается номер удаляемой страницы в **памяти**;
4. Организовывать циклическое обращение к страницам размещенным в памяти по нажатию на кнопку. При этом случайным образом задается количество обращений к страницам (диапазон 1..10). Для каждого обращения генерируется, случайным образом, номер страницы из диапазона [0; количество страниц памяти]. При обращении к странице в зависимости, от варианта, увеличивается ее внутренний счетчик обращений или устанавливается флаг обращения

Вариант	Задание
4	**Глобальное размещение**. Алгоритм замещения – LRU CLOCK. Существует глобальный счетчик обращений к страницам. При каждом обращении к станице в ее внутренний регистр заносится значение глобального счетчика. Выгружается страница с наименьшим значением счетчика.

Листинг программы:

```
#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

#include <conio.h>

#include <iostream>

#include <queue>

using namespace std;


int id=0;

int metka=0;

struct page

{
```

```
        int status;
        double memory;
        int processid;
        int counter;
        double badmemory;
};

struct process
{
        int id;
        int memory;
};

void vivod(page *massiv,int kolvo);
double allbadmemory(page *massiv,int kolvo);
void mainprint(page *massiv,double allmemory,int kolvo);
int svoboda(page *massiv,int kolvo);
void deletepage(page *massiv,int index);
void deleteprocess(page *massiv,int kolvo);
void add(page *massiv,int kolvo,process tmp,double gig);


int main()
{
        int i,kolvo,control;
```

```
setlocale(LC_ALL,"Rus");
process tmp;
double allmemory,gig;
page *massiv = new page[kolvo];
printf("Введите кол-во памяти:\n");
scanf("%lf",&allmemory);
printf("Введите кол-во страниц:\n");
scanf("%d",&kolvo);
system("cls");
gig = allmemory/kolvo;
for(i=0;i<kolvo;i++)
{
        massiv[i].memory=allmemory/kolvo;
        massiv[i].status=0;
        massiv[i].counter=0;
        massiv[i].badmemory=0;
        massiv[i].processid=-1;
}
do
{
        mainprint(massiv,allmemory,kolvo);
   vivod(massiv,kolvo);
   printf("1.Добавить процесс\n");
   printf("2.Удалить заданную страницу\n");
   printf("3.Удалить заданный процесс\n");
```

```
printf("4.Циклическое обращение к страницам\n");
printf("0.Выход\n");
    scanf("%d",&control);
    switch(control)
    {
          case 1:
          {
                printf("Введите память процесса\n");
                scanf("%d",&tmp.memory);
                add(massiv,kolvo,tmp,gig);
                printf("\n");
                mainprint(massiv,allmemory,kolvo);
                printf("\n");
        vivod(massiv,kolvo);
                break;
          }
          case 2:
          {
                int index;
                printf("Введите номер страницы для удаления\n");
                scanf("%d",&index);
                deletepage(massiv,index);
                printf("\n");
                mainprint(massiv,allmemory,kolvo);
                printf("\n");
```

```
vivod(massiv,kolvo);
            break;
   }
   case 3:
   {
            deleteprocess(massiv,kolvo);
            printf("\n");
            mainprint(massiv,allmemory,kolvo);
            printf("\n");
vivod(massiv,kolvo);
            break;
   }
   case 4:
   {
            int rand1 =rand()%10+1;
            printf("\nКоличество обращений %d\n",rand1);
            int kolich=0;
            while(kolich!=rand1)
            {
                     massiv[rand() % kolvo].counter++;
                     kolich++;
            }
            printf("\n");
            mainprint(massiv,allmemory,kolvo);
            printf("\n");
```

```
                    vivod(massiv,kolvo);
                        break;
                    }
                    case 0:return(0);

                }
        }while(control!=0);

        getch();
        return(0);
}

void vivod(page *massiv,int kolvo)
{
        for(int i=0;i<kolvo;i++)
        {
                printf("Страница №%d  ",i+1);
                printf("Счетчик обращений %d  ",massiv[i].counter);
                printf("%.1lf  ",massiv[i].badmemory);
                if(massiv[i].processid==-1)printf("Страница не занята\n");
                else printf("Id процесса :%d\n",massiv[i].processid);


        }
}
```

```
double allbadmemory(page *massiv,int kolvo)
{
	int i;
	double allbadmemorycount1;
	for(i=0;i<kolvo;i++)allbadmemorycount1+=massiv[i].badmemory;
	printf("Объём заполненной памяти: %lf\n",allbadmemorycount1);
	return(allbadmemorycount1);
}

void mainprint(page *massiv,double allmemory,int kolvo)
{
	int i,fig=0;
	for(i=0;i<kolvo;i++)
	{
		if(massiv[i].status==0)fig++;

	}
	printf("__________Стата_____________\n");
	printf("Объем свободной памяти :%.1lf\n",allmemory-
allbadmemory(massiv,kolvo));
	printf("Число свободных страниц:%d\n",fig);
	printf("Размер страницы :%.1lf\n",allmemory/kolvo);
	printf("_____________________________\n\n\n");
}
int svoboda(page *massiv,int kolvo)
{
```

```
	int vern=0,i;
	for(i=0;i<kolvo;i++)
	{
		if(massiv[i].status==0)vern++;
	}
	return vern;
}


void deletepage(page *massiv,int index)
{

	massiv[index-1].status=0;
	massiv[index-1].badmemory=0;
	massiv[index-1].processid=-1;
	massiv[index-1].counter++;
}
void deleteprocess(page *massiv,int kolvo)
{
	int index,i;
	printf("Введите id процесса\n");
	scanf("%d",&index);
	for(i=0;i<kolvo;i++)
	{
		if(massiv[i].processid==index)
```

```
                {
                        massiv[i].status=0;
                        massiv[i].badmemory=0;
                        massiv[i].processid=-1;
                        massiv[i].counter++;
                }
        }
}
void add(page *massiv,int kolvo,process tmp,double gig)
{
        int i;
        double pam = gig;
        if(tmp.memory>pam)
        {
                while(tmp.memory!=0)
                {
                        if(svoboda(massiv,kolvo)>0)
                        {
                                for(int i=0;i<kolvo;i++)
                                        {
                                    if(massiv[i].status==0)
                                     {

        if(tmp.memory>massiv[i].memory)
                                                        {
                                                                massiv[i].status=1;
```

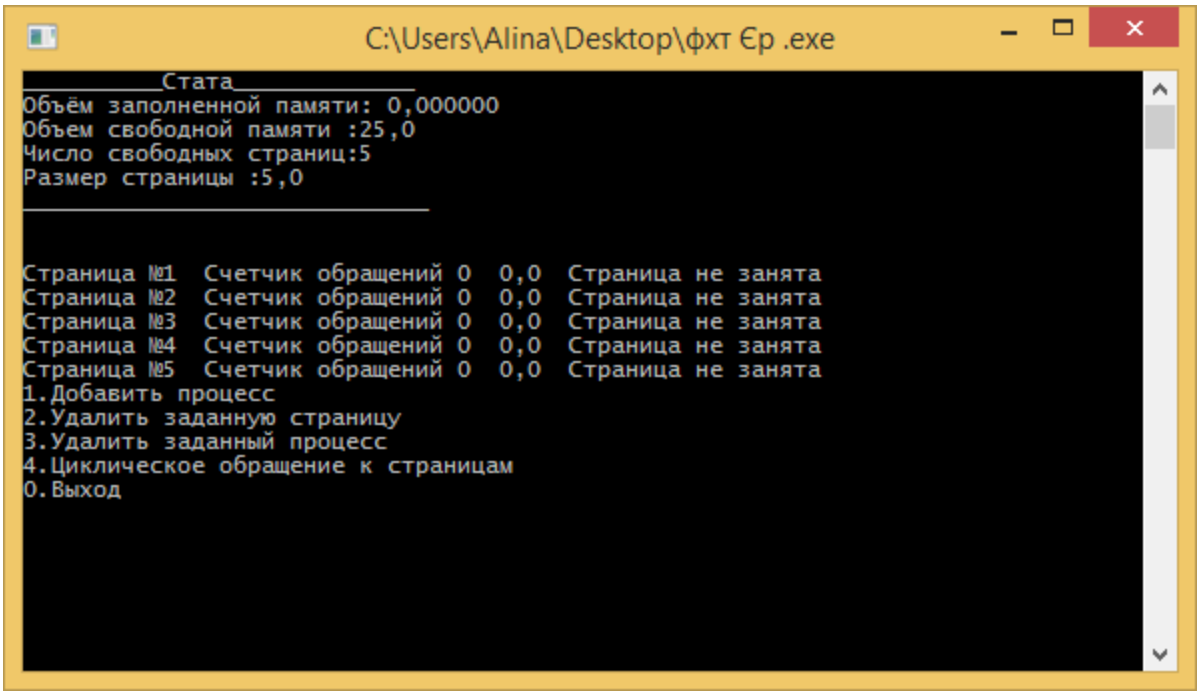
```
massiv[i].badmemory=massiv[i].memory;
                                        massiv[i].processid=id;
                                        massiv[i].counter++;
                                        tmp.memory-
=massiv[i].memory;
                                        break;
                                    }
                                    else
                                    {
                                        massiv[i].status=1;

massiv[i].badmemory=tmp.memory;
                                        massiv[i].processid=id;
                                        massiv[i].counter++;
                                        tmp.memory=0;
                                        break;
                                    }
                                }
                            }
                        }
                        else
                        {
                            if(tmp.memory>pam)
                        {
                            int  j;
```

```
                int er=0;
                int min=massiv[0].counter;
for(j=1;j<kolvo;j++)
        {
        if(massiv[j].counter<min) {
        min=massiv[j].counter;
        er=j;}}
                deletepage(massiv,er+1);
                massiv[er].status=1;
                massiv[er].badmemory=massiv[i].memory;
                massiv[er].processid=id;
                tmp.memory-=massiv[er].memory;


        }
        else
        {
                int er=0,j;
                int min=massiv[0].counter;
for(j=1;j<kolvo;j++)
        {
        if(massiv[j].counter<min) {
        min=massiv[j].counter;
        er=j;}}
                deletepage(massiv,er+1);
                massiv[er].status=1;
```

```
                massiv[er].badmemory=tmp.memory;
                massiv[er].processid=id;
                tmp.memory=0;

            }
            }
        }
        id++;
    }
    else
{
    if(svoboda(massiv,kolvo)>0)
    {
        for(int i=0;i<kolvo;i++)
        {
            if(massiv[i].status==0)
            {
                massiv[i].status=1;
                massiv[i].badmemory=tmp.memory;
                massiv[i].processid=id;
                massiv[i].counter++;
                //q.push(i);
                id++;
                break;
            }
```

```
                }
        }
        else
        {
                int er=9999,j,min;// = q.front();
                min=massiv[0].counter;
                er=0;
                for(j=1;j<kolvo;j++)
                        {
                        if(massiv[j].counter<min) {
                        min=massiv[j].counter;
                        er=j;}}
                deletepage(massiv,er+1);
                //printf("3.%d\n\n\n",er);
                //q.pop();
                //q.push(er);
                massiv[er].status=1;
                massiv[er].badmemory=tmp.memory;
                massiv[er].processid=id;
                id++;
        }
    }
    }
```

Результат выполнения программы:



Вывод: Были рассмотрены простейшие алгоритмы для работы с памятью