

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Ульяновский государственный технический университет»
Кафедра «Вычислительная техника им. П. И. Соснина»

Операционные системы

(название дисциплины)

Лабораторная работа №3

«Менеджер виртуальной памяти»

(название (тема) работы)

Выполнил:
студент группы ИВТАПбд-31
Молчанов А. В.
(Фамилия И. О.)

Проверил:
преподаватель, аспирант кафедры «ВТ»
(должность)
Беляев К. С.
(Фамилия И. О.)

Ульяновск

2023

Постановка задачи

Данная лабораторная работа заключается в написании программы, которая должна считать входной файл `addresses.txt` и, используя TLB и таблицу страниц, преобразовать каждый логический адрес из него в соответствующий ему физический адрес для виртуального адресного пространства размером 32 768 байт (2^8 в степени 15). Это также будет включать в себя устранение ошибок страниц с использованием подкачки по запросу (`demand paging`), управление TLB и реализацию алгоритма замещения страниц.

Спецификация:

- Записей в таблице страниц: 256 (2^8 в степени 8);
- Размер одной страницы: 256 байт;
- Записей в TLB: 16;
- Фреймов в физической памяти: 128 (2^7 в степени 7);
- Размер одного фрейма: 256 байт.

По завершению своей работы, программа должна вывести на экран следующую информацию:

1. Транслируемый логический адрес (целочисленное значение, которое считывается из файла `addresses.txt`);
2. Соответствующий физический адрес (во что программа преобразует логический адрес из предыдущего пункта);
3. Значение байта со знаком (`signed`), хранящееся в виртуальном адресном пространстве по преобразованному физическому адресу;
4. Частота ошибок страниц – процент ссылок на адреса, которые привели к ошибкам страниц;
5. Частота попаданий в TLB – процент ссылок на адреса, которые были найдены в TLB.

Детали реализации

Программа начинается с объявления некоторых константных переменных байтной размерности для таких глобальных структур данных как таблица страниц, виртуальное адресное пространство и TLB (Рисунок 1).

```
C main.c x
C main.c > ...
1  #include "stdbool.h"
2  #include "stdio.h"
3  #include "stdlib.h"
4  #include "string.h"
5  #include "sys/types.h"
6
7  #define PAGE_SIZE 256 // 2^8 байт
8  #define PAGE_TABLE_SIZE 256 //
9  #define FRAME_TABLE_SIZE 128
10 #define TLB_SIZE 16
11
12 // Структура таблицы страниц
13 struct page_table_entry
14 {
15     int frame_number;
16 } page_table[PAGE_TABLE_SIZE];
17
18 // Структура виртуального адресного пространства (физической памяти)
19 struct frame_table_entry
20 {
21     struct frame_entry
22     {
23         char page[PAGE_SIZE];
24         int usage_time; // Целое число, характеризующее последнее время обращения к записи (чем оно больше, тем позднее время)
25     } frame;
26 } physical_memory[FRAME_TABLE_SIZE];
27
28 // Структура TLB
29 struct TLB_entry
30 {
31     int page_number;
32     int frame_number;
33     int usage_time;
34 } TLB[TLB_SIZE];
35
36 #define PAGE_NUMBER_MASK 0xFFFFFFF0 // ~ 2^32 = 4294967040 = 11111111111111 1111111 00000000 (номер страницы занимает в логическом адресе средние 8 бит)
37 #define OFFSET_MASK 0x000000FF // 2^8 - 1 = 255 = 0000000000000000 00000000 11111111 (смещение занимает в логическом адресе крайние правые 8 бит)
38
```

Рисунок 1. Фрагмент кода №1

Константы масок PAGE_NUMBER_MASK и OFFSET_MASK представляют собой 32-разрядные целые числа. Такую же разрядность имеют и логические адреса из файла addresses.txt, которые будут маскироваться двумя переменными выше для выделения из них номера страницы и смещения в ней (Рисунок 2).

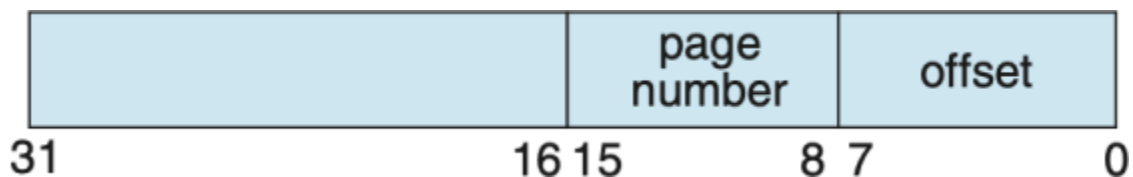
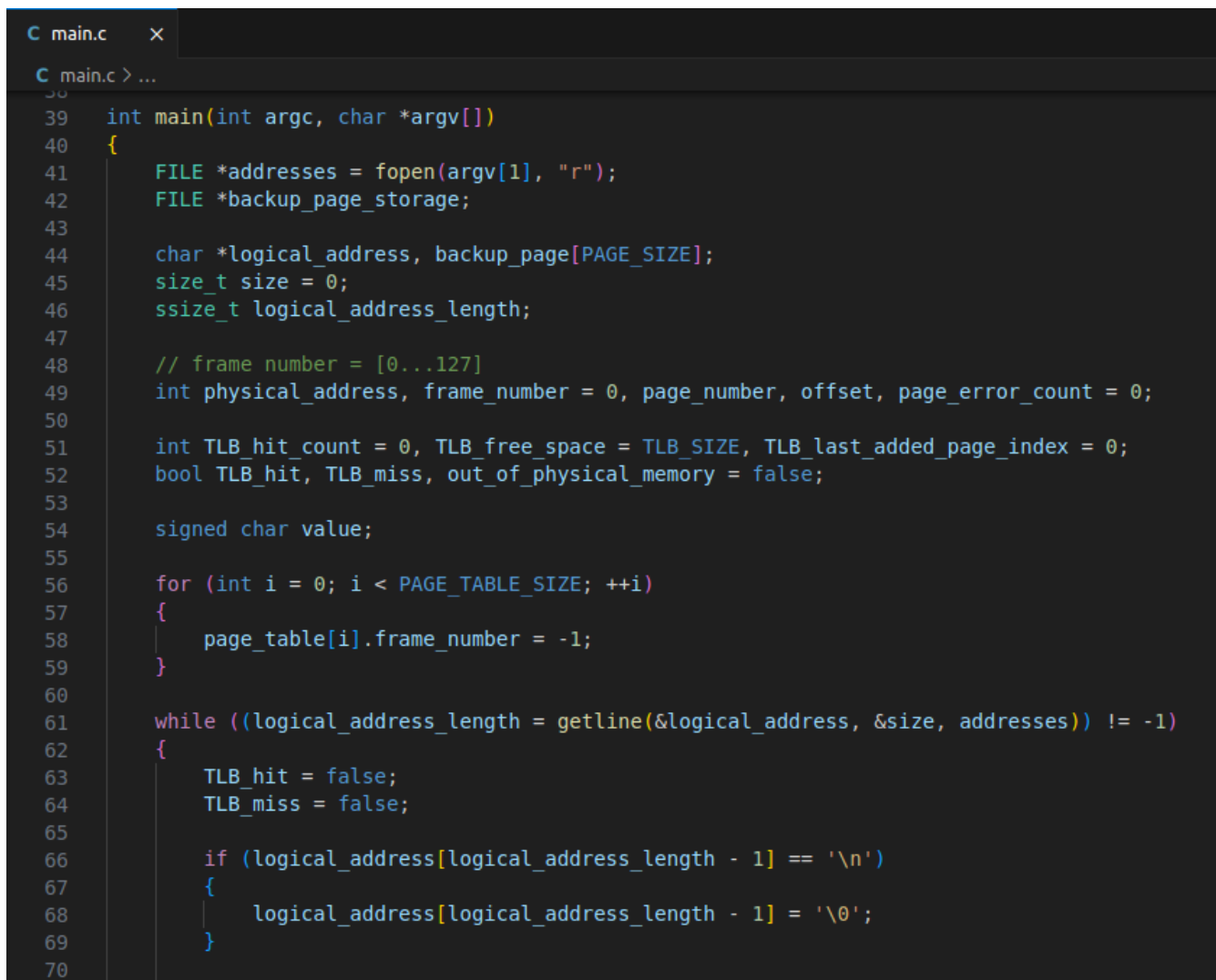


Рисунок 2. Представление логического адреса

Перед непосредственным считыванием и обработкой логических адресов, в единственной в данной программе функции `main` объявляется множество вспомогательных переменных, начиная файлами с самими адресами и резервным хранилищем страниц, заканчивая извлекаемым из физической памяти значением по подписанному байту (Рисунок 3).



```
C main.c x
C main.c > ...
38
39 int main(int argc, char *argv[])
40 {
41     FILE *addresses = fopen(argv[1], "r");
42     FILE *backup_page_storage;
43
44     char *logical_address, backup_page[PAGE_SIZE];
45     size_t size = 0;
46     ssize_t logical_address_length;
47
48     // frame number = [0...127]
49     int physical_address, frame_number = 0, page_number, offset, page_error_count = 0;
50
51     int TLB_hit_count = 0, TLB_free_space = TLB_SIZE, TLB_last_added_page_index = 0;
52     bool TLB_hit, TLB_miss, out_of_physical_memory = false;
53
54     signed char value;
55
56     for (int i = 0; i < PAGE_TABLE_SIZE; ++i)
57     {
58         page_table[i].frame_number = -1;
59     }
60
61     while ((logical_address_length = getline(&logical_address, &size, addresses)) != -1)
62     {
63         TLB_hit = false;
64         TLB_miss = false;
65
66         if (logical_address[logical_address_length - 1] == '\n')
67         {
68             logical_address[logical_address_length - 1] = '\0';
69         }
70
```

Рисунок 3. Фрагмент кода №2

Считывая в цикле while из файла addresses.txt по одному логическому адресу, операциями побитового сдвига вправо и умножения из него извлекается сначала номер страницы, а затем смещение в ней (Рисунки 2, 4), после чего происходит обращение к буферу ассоциативной трансляции (TLB) и попытка найти в нём замаскированный номер страницы. В случае попадания (hit) в TLB, из него извлекается номер фрейма виртуального адресного пространства и путём его побитового сдвига уже влево, а также прибавления смещения, формируется физический адрес и значение подписанного по нему байта (Рисунок 4).

```
page_number = (atoi(logical_address) & PAGE_NUMBER_MASK) >> 8; // >> 8 - сдвиг на 8 бит вправо для получения числа без обнулённых цифр с правого края
offset = atoi(logical_address) & OFFSET_MASK; // & - операция побитового И

// Обращение к TLB (Translation Lookaside Buffer - Буфер ассоциативной трансляции)
for (int i = 0; i < TLB_SIZE; ++i)
{
    TLB[i].usage_time += 1;

    if (!TLB_hit)
    {
        if (TLB[i].page_number == page_number)
        {
            TLB_hit = true;
            TLB_miss = false;

            TLB[i].usage_time = 0; // 0 - к записи только что произошло обращение
            physical_memory[TLB[i].frame_number].frame.usage_time = 0;

            physical_address = TLB[i].frame_number << 8 | offset; // << 8 для получения шестнадцатитрёхбитного адреса
            value = physical_memory[TLB[i].frame_number].frame.page[offset];

            ++TLB_hit_count;
        }
        else
        {
            TLB_miss = true;
            TLB_hit = false;
        }
    }
}
```

Рисунок 4. Фрагмент кода №3

Если же обращение к буферу не дало никакого результата (произошёл промах – miss), то производится обращение уже к другой структуре данных – таблице страниц и попытка извлечь номер фрейма из неё с последующими операциями над ним в случае успеха.

В случае же очередной неудачи, приходится рассчитывать на последнее – резервное хранилище страниц, представленное двоичным (бинарным) файлом BACKING_STORE.bin.

Обращение и последующая работа с данными, извлечёнными из данного файла, представлены и подробно описаны в комментариях к программному коду на Рисунке 5 ниже.

```
if (TLB_miss)
{
    if (page_table[page_number].frame_number != -1)
    {
        physical_address = page_table[page_number].frame_number << 8 | offset;
    }
    else
    {
        ++page_error_count;

        backup_page_storage = fopen("BACKING_STORE.bin", "r");
        int backup_page_number = 0;

        // Считываем из резервного хранилища страниц одну 256 байтную страницу
        while (fread(backup_page, PAGE_SIZE, 1, backup_page_storage))
        {
            // Если номер считанной страницы совпал с номером страницы, выдавшей ошибку
            if (backup_page_number == page_number)
            {
                // Если все 128 фреймов физической памяти заполнены страницами (отсутствует свободная физическая память)
                if (frame_number > 127 || out_of_physical_memory)
                {
                    out_of_physical_memory = true;

                    int maximum_usage_time = 0;
                    int LRU_frame_index = 0;

                    // Ищем самый неиспользуемый (LRU - Least Recently Used) фрейм
                    for (int i = 0; i < FRAME_TABLE_SIZE; ++i)
                    {
                        if (physical_memory[i].frame.usage_time > maximum_usage_time)
                        {
                            maximum_usage_time = physical_memory[i].frame.usage_time;
                            LRU_frame_index = i;
                        }
                    }

                    // Перезаписываем его новыми данными
                    memcpy(physical_memory[LRU_frame_index].frame.page, backup_page, PAGE_SIZE);
                    physical_memory[LRU_frame_index].frame.usage_time = 0;

                    frame_number = LRU_frame_index;
                }
                else
                {
                    // Сохраняем страницу из хранилища, а также последнее время обращения к ней в свободном фрейме физической памяти
                    memcpy(physical_memory[frame_number].frame.page, backup_page, PAGE_SIZE);
                    physical_memory[frame_number].frame.usage_time = 0;
                }
            }
        }
    }
}
```

Рисунок 5. Фрагмент кода №4

После чего появляется необходимость в обновлении как таблицы страниц, так и Translation Lookaside Buffer по политике наиболее неиспользуемой записи – LRU (Least Recently Used) и, в конечном итоге, составление из полученной информации заветного физического адреса (Рисунок 6).

```

// Обновление таблицы страниц
page_table[page_number].frame_number = frame_number;

// Обновление TLB
if (TLB_free_space != 0) // Если в TLB есть свободное место
{
    --TLB_free_space;

    // Последовательно добавляем в него записи
    TLB[TLB_last_added_page_index].page_number = page_number;
    TLB[TLB_last_added_page_index].frame_number = frame_number;
    TLB[TLB_last_added_page_index].usage_time = 0;

    ++TLB_last_added_page_index;
}
else
{
    int maximum_usage_time = 0;
    int LRU_page_index = 0;

    // Если в TLB нет свободного места, то ищем в нём LRU-запись
    for (int i = 0; i < TLB_SIZE; ++i)
    {
        if (TLB[i].usage_time > maximum_usage_time)
        {
            maximum_usage_time = TLB[i].usage_time;
            LRU_page_index = i;
        }

        // Перезаписываем её новыми данными
        TLB[LRU_page_index].page_number = page_number;
        TLB[LRU_page_index].frame_number = frame_number;
        TLB[LRU_page_index].usage_time = 0;
    }

    physical_address = frame_number << 8 | offset;

    if (!out_of_physical_memory)
    {
        ++frame_number;
    }

    break;
}
else
{
    ++backup_page_number;
}

```

Рисунок 6. Фрагмент кода №5

Завершается цикл считывания логических адресов, функция main и вся программа в целом выводом необходимой по каждому адресу информации и статистики (частота ошибок страниц и частота попаданий в TLB) на экран (Рисунок 7).

```
        value = physical_memory[page_table[page_number].frame_number].frame.page[offset];
        physical_memory[page_table[page_number].frame_number].frame.usage_time = 0;
    }

    printf("Virtual address: %s Physical address: %d Value: %d\n", logical_address, physical_address, value);

    for (int i = 0; i < FRAME_TABLE_SIZE; ++i)
    {
        physical_memory[i].frame.usage_time += 1;
    }
}

printf("\nPage error rate = %.f%%\n", page_error_count / 1000.0 * 100);
printf("TLB hit rate = %.f%%\n", TLB_hit_count / 1000.0 * 100);

fclose(addresses);
fclose(backup_page_storage);

return 0;
```

Рисунок 7. Фрагмент кода №6

Результат работы программы

На Рисунке 8 можно заметить, что выходной файл output.txt работы программы начинает местами не совпадать с образцовым выводом из файла correct.txt немного дальше 128 строки (171).

Это связано с реализацией дополнительного задания замещения страниц, при котором физическая память имеет размер в два раза меньший, чем виртуальная и состоит в таком случае из 128 исходных страничных фреймов вместо 256, а это значит, что при формировании физических адресов используются числа от 0 до 127 включительно, вместо 255 для которых представлен файл correct.txt.

Что происходит при заполнении всех 128 исходно свободных фреймов виртуального адресного пространства можно наблюдать на Рисунке 5 выше.

Virtual address	Physical address	Value
15555	31939	48
47475	32115	92
15328	32488	0
34621	17981	0
51365	32677	0
32820	4148	0
48855	215	-75
12224	2752	0
2035	755	-4
60539	7291	30
14595	771	64
13853	31261	0
24143	33015	-109
15216	32368	0
8113	3969	0
22640	2928	0
32978	4306	32
39151	4079	59
19520	22592	0
58141	2333	0
63959	21975	117
52040	34352	0
55842	4386	54
585	17225	0
51229	32541	0
64181	4533	0
54879	3879	41
28210	4658	27
10268	14620	0
15395	31779	8
12884	30292	0
2149	35173	0
53483	35563	58
59606	26070	58
14981	24769	0
36872	36648	0
23197	35997	0
36518	14502	35
13361	36145	0
19810	36450	19
25955	36707	88
62678	470	61
26821	36773	0
29409	37089	0
38111	37343	55
58573	15505	0
56840	37384	0
41306	24922	40
54426	19354	53
3617	18017	0
50652	18652	0
41452	25068	0

Рисунок 8. Сравнение выходных файлов output.txt и correct.txt командой vimdiff