



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

*«Мониторинг состояния физических
страниц, выделенных процессу»*

Студент ИУ7-73Б
(Группа)

(Подпись, дата)

Р. Р. Хамзина
(И. О. Фамилия)

Руководитель курсовой работы

(Подпись, дата)

Н. Ю. Рязанова
(И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 Аналитический раздел	5
1.1 Постановка задачи	5
1.2 Анализ управления памятью	5
1.3 Анализ структур ядра, предоставляющих информацию о страницах, выделенных процессу	7
1.4 Анализ использования записей таблицы страниц	10
2 Конструкторский раздел	12
2.1 Последовательность преобразований	12
2.2 Алгоритм сканирования виртуальных страниц	12
2.3 Алгоритм обращения к дескриптору физической страницы	13
2.4 Алгоритм получения информации о состоянии физической страницы	15
2.5 Структура программного обеспечения	15
3 Технологический раздел	16
3.1 Выбор языка и среды программирования	16
3.2 Функции загрузки и выгрузки модуля	16
3.3 Функция сканирования виртуальных страниц	17
3.4 Функция обращения к дескриптору физической страницы	18
3.5 Функция получения информации о состоянии физической страницы	19
4 Исследовательский раздел	20
4.1 Примеры работы разработанного ПО	20
ЗАКЛЮЧЕНИЕ	22
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	23
ПРИЛОЖЕНИЕ А Исходный код загружаемого модуля	24

ВВЕДЕНИЕ

Основной задачей операционной системы является управление процессами и выделение процессам ресурсов. Ядро операционной системы Linux рассматривает физические страницы как основную единицу управления памятью.

Важно определить состояние страницы, так как необходимо знать, какая из страниц свободна. Если страница занята, то ядро должно получить информацию о ее владельце. К возможным владельцам относятся пользовательские процессы.

Данная курсовая работа посвящена мониторингу состояния физических страниц, выделенных процессу.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра для мониторинга состояния физических страниц, выделенных процессу.

Для решения поставленной задачи необходимо:

- проанализировать структуры и функции ядра, предоставляющие информацию о физических страницах, выделенных процессу;
- разработать алгоритмы и структуру программного обеспечения;
- реализовать программное обеспечение;
- провести анализ работы разработанного программного обеспечения.

1.2 Анализ управления памятью

Операционная система Linux является системой с поддержкой виртуальной памяти [1]. Для каждого процесса создается адресное пространство, которое делится на блоки равного размера, называемые страницами. Размер страницы определяется системно. Единицей деления физической памяти является физическая страница или страничный кадр. При таком механизме организации памяти каждой виртуальной странице ставится в соответствие физический адрес.

Для отображения виртуальной памяти в физическую используются таблицы страниц. Виртуальный адрес разбивается на части: индексы и смещение. Информация, хранящаяся в элементе таблицы страниц, соответствующем заданному индексу, указывает либо на адрес другой таблицы, либо на адрес физической страницы памяти, в которой хранятся нужные данные. Схема преобразования виртуального адреса в физический адрес показана на рисунке 1.1.

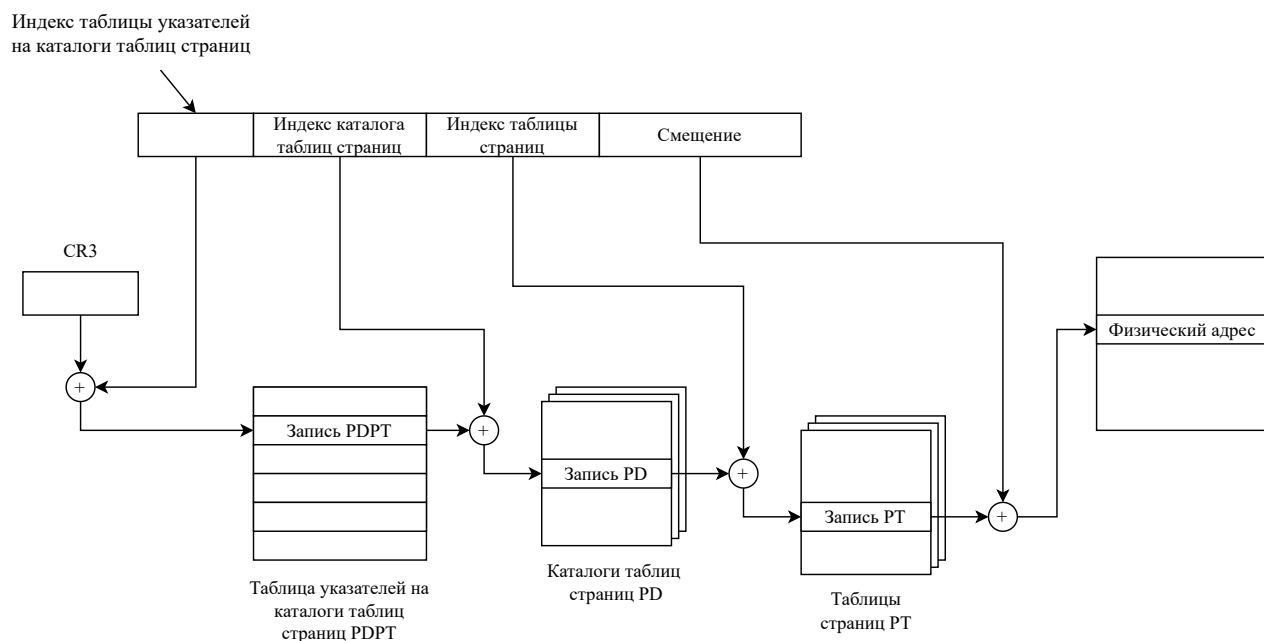


Рисунок 1.1 – Преобразование виртуального адреса в физический адрес

Для каждого процесса создается адресное пространство, которое описывается структурой, называемой дескриптором памяти. В поле **pgd** дескриптора памяти хранится указатель на глобальный каталог страниц текущего процесса PGD. Элементы в таблице PGD содержат указатели на каталоги страниц среднего уровня PMD. Элементы таблиц PMD содержат указатели на таблицы PTE, записи которой указывают на страницы физической памяти. Связь таблиц страниц представлена на рисунке 1.2.

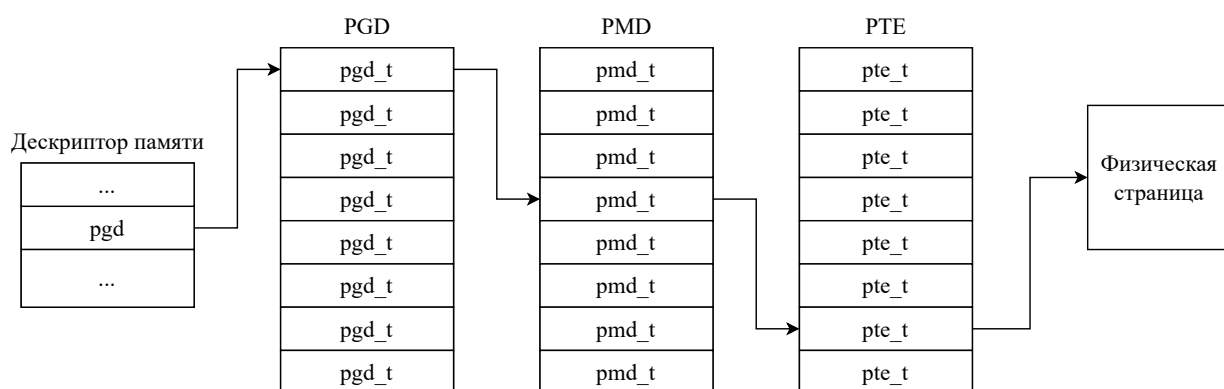


Рисунок 1.2 – Связь таблиц страниц

1.3 Анализ структур ядра, предоставляющих информацию о страницах, выделенных процессу

Для описания процесса в ядре используется структура `task_struct`, которая называется дескриптором процесса. Структура описана в файле `<linux/sched.h>`. Дескриптор процесса позволяет получить информацию о состоянии процесса, открытых файлах и другом. Некоторые поля этой структуры приведены в листинге 1.1.

Листинг 1.1 – Структура `task_struct`

```
1 struct task_struct {
2     ...
3     unsigned int      __state;
4     ...
5     unsigned int      flags;
6     ...
7     int               prio;
8     int               static_prio;
9     ...
10    struct list_head    tasks;
11    ...
12    struct mm_struct    *mm;
13    struct mm_struct    *active_mm;
14    ...
15    int                exit_code;
16    ...
17    pid_t              pid;
18    pid_t              tgid;
19    ...
20    char               comm[TASK_COMM_LEN];
21    ...
22    struct fs_struct    *fs;
23    struct files_struct *files;
24    ...
25 };
```

У объекта `task_struct` есть поле `mm`, содержащее указатель на структуру `mm_struct` — дескриптор памяти процесса. Эта структура описана в файле `<linux/mm_types.h>`. Дескриптор памяти процесса предоставляет всю информацию, относящуюся к адресному пространству процесса. Некоторые поля этой структуры показаны в листинге 1.2.

Листинг 1.2 – Структура mm_struct

```
1 struct mm_struct {
2     struct vm_area_struct *mmap;
3     struct rb_root mm_rb;
4     ...
5     unsigned long mmap_base;
6     ...
7     unsigned long task_size;
8     unsigned long highest_vm_end;
9     pgd_t * pgd;
10    ...
11    atomic_t mm_users;
12    atomic_t mm_count;
13    ...
14    int map_count;
15    spinlock_t page_table_lock;
16    struct rw_semaphore mmap_lock;
17    struct list_head mmlist;
18    ...
19    unsigned long total_vm;
20    unsigned long locked_vm;
21    ...
22    unsigned long start_code, end_code, start_data, end_data;
23    unsigned long start_brk, brk, start_stack;
24    unsigned long arg_start, arg_end, env_start, env_end;
25    ...
26    struct mm_rss_stat rss_stat;
27    ...
28    unsigned long flags;
29    ...
30 };
```

Поле `mmap` объекта `mm_struct` содержит указатель на структуру `vm_area_struct`, которая определена в файле `<linux/mm_types.h>`. Данная структура используется для описания одной непрерывной области памяти в данном адресном пространстве. Некоторые поля этой структуры представлены в листинге 1.3.

Листинг 1.3 – Структура vm_area_struct

```
1 struct vm_area_struct {
2     unsigned long vm_start;
3     unsigned long vm_end;
4     struct vm_area_struct *vm_next, *vm_prev;
5     ...
6     struct mm_struct *vm_mm;
7     ...
8     unsigned long vm_flags;
```

```

9      ...
10     struct list_head anon_vma_chain;
11     struct anon_vma *anon_vma;
12     const struct vm_operations_struct *vm_ops;
13     unsigned long vm_pgoff;
14     struct file * vm_file;
15     void * vm_private_data;
16     ...
17 };

```

Каждый дескриптор памяти связан с уникальным диапазоном адресов в адресном пространстве процесса. В поле `vm_start` `vm_area_struct` хранится начальный адрес этого диапазона, а в поле `vm_end` — адрес первого байта, расположенного после описываемого диапазона.

Для описания физической страницы памяти в ядре используется структура `page`, которая определена в файле `<linux/mm_types.h>`. Некоторые поля этой структуры приведены в листинге 1.4.

Листинг 1.4 – Структура `page`

```

1 struct page {
2     unsigned long flags;
3     struct list_head lru;
4     struct address_space *mapping;
5     pgoff_t index;
6     unsigned long private;
7     ...
8     struct list_head slab_list;
9     struct page *next;
10    int pages;
11    int pobjects;
12    ...
13    void *freelist;
14    ...
15    atomic_t _mapcount;
16    unsigned int page_type;
17    unsigned int active;
18    int units;
19    atomic_t _refcount;
20    ...
21    void *virtual;
22    ...
23 };

```

Для страниц, которые можно сопоставить с пользовательским пространством, в поле `_mapcount` объекта `page` хранится количество записей в таблице

страниц, указывающих на данную страницу [2].

В поле `_refcount` объекта `page` хранится счетчик использования страницы, отражающий количество ссылок в системе на эту страницу. Только что выделенная страница имеет счетчик ссылок, равный 1. Когда счетчик ссылок достигает нуля, страница освобождается.

Для определения счетчика ссылок на страницу используется функция ядра `page_ref_count`, определенная в `<linux/page_ref.h>` (листинг 1.5):

Листинг 1.5 – Определение счетчика ссылок на страницу

```
1 #include <linux/atomic.h>
2 #include <linux/mm_types.h>
3 #include <linux/page-flags.h>
4
5 static inline int page_ref_count(const struct page *page)
6 {
7     return atomic_read(&page->_refcount);
8 }
```

1.4 Анализ использования записей таблицы страниц

Для доступа к дескриптору физической страницы используются записи таблицы страниц [3]. Для перемещения по таблицам страниц в файле `<asm/pgtable.h>` определены следующие функции ядра (листинг 1.6):

Листинг 1.6 – Перемещение по таблицам страниц

```
1 #define pte_offset_map(dir, addr)    pte_offset_kernel((dir), (addr))
2 #define pte_page(pte)                (mem_map+pte_pagenr(pte))
3 #define pte_pagenr(pte)              ((__pte_page(pte) - PAGE_OFFSET) >>
4     PAGE_SHIFT)
5
6 pgd_t *pgd_offset(const struct mm_struct *mm, unsigned long address);
7 pmd_t *pmd_offset(pgd_t *dir, unsigned long address);
8 pte_t *pte_offset_kernel(pmd_t *dir, unsigned long address);
9 unsigned long __pte_page(pte_t pte);
```

Доступ к физической странице осуществляется следующим образом:

1. Системный вызов `pgd_offset()` принимает дескриптор памяти процесса и виртуальный адрес и возвращает элемент глобального каталога страниц PGD текущего процесса.
2. Системный вызов `pmd_offset()` запись глобального каталога страниц PGD и виртуальный адрес и возвращает запись каталога страниц среднего

уровня PMD.

3. Системный вызов `pte_offset_map()` принимает элемент каталога страниц среднего уровня PMD и виртуальный адрес и возвращает запись таблицы PTE.
4. Системный вызов `pte_page()` по полученной записи таблицы PTE возвращает объект `page`.

Функции ядра, представленные в листинге 1.7, используются для определения того, существуют ли соответствующие записи таблицы страниц. В случае отсутствия записи в таблице возвращается 1.

Листинг 1.7 – Определения счетчика ссылок на страницу

```
1 int pgd_none(pgd_t pgd);  
2 int pmd_none(pmd_t pmd);  
3 int pte_none(pte_t pte);
```

Функции ядра, показанные в листинге 1.8, используются для проверки записей при передаче в качестве входных параметров функциям, которые могут изменить значение записей.

Листинг 1.8 – Определения счетчика ссылок на страницу

```
1 int pgd_bad(pgd_t pgd);  
2 int pmd_bad(pmd_t pmd);
```

Вывод

В соответствии с проведенным анализом для решения поставленной задачи необходимо разработать алгоритм сканирования виртуальных страниц и алгоритм обращения к дескриптору физической страницы. Для доступа к дескриптору физической страницы следует использовать глобальный каталог страниц PGD, каталог страниц среднего уровня PMD и таблицу страниц PTE процесса. Для доступа к таблицам страниц процесса необходимо обращаться к следующим структурам ядра: `task_struct`, `mm_struct`, `vm_area_struct`. Для получения информации о состоянии физической страницы следует обращаться к полям структуры `page`.

2 Конструкторский раздел

2.1 Последовательность преобразований

На рисунке 2.1 приведена диаграмма состояний IDEF0 нулевого уровня, а на рисунке 2.2 — диаграмма состояний IDEF0 первого уровня.

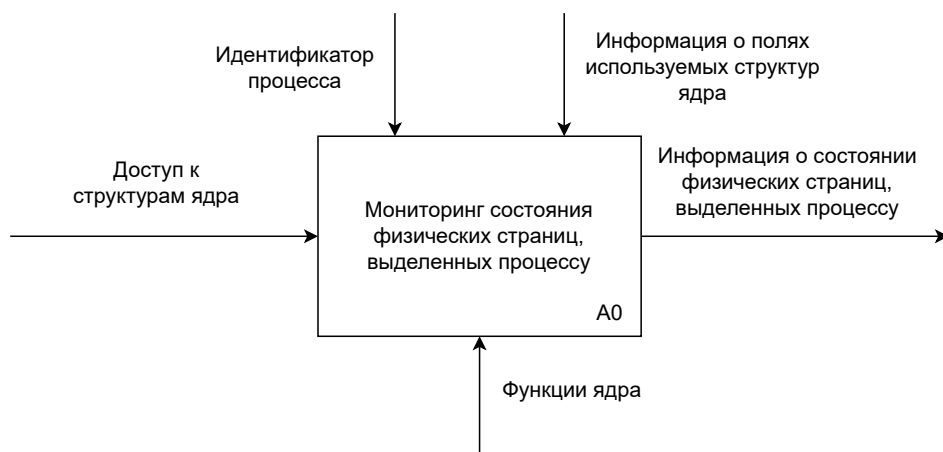


Рисунок 2.1 – Диаграмма состояний IDEF0 нулевого уровня

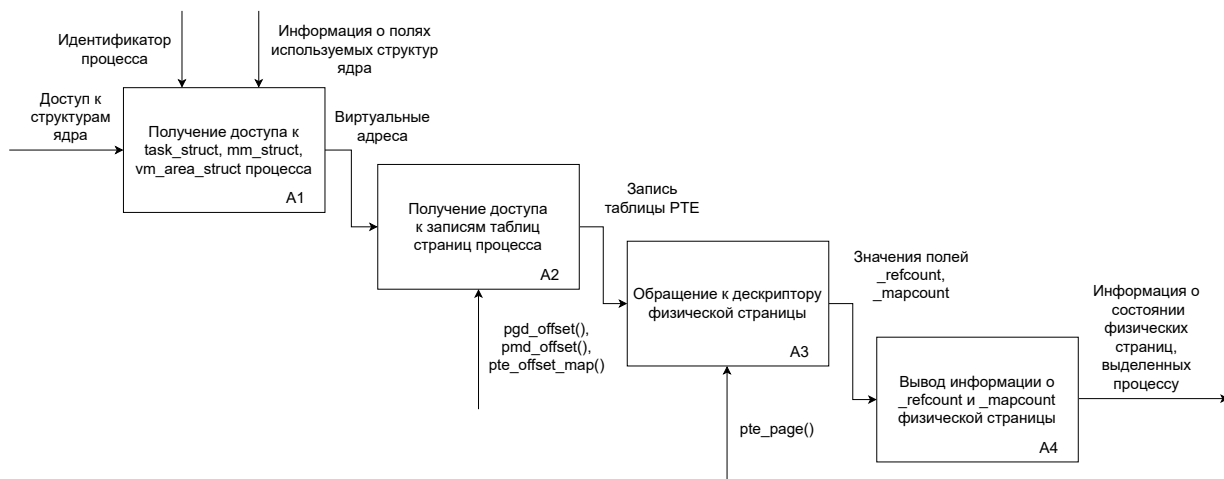


Рисунок 2.2 – Диаграмма состояний IDEF0 первого уровня

2.2 Алгоритм сканирования виртуальных страниц

На рисунке 2.3 представлена схема алгоритма сканирования виртуальных страниц.

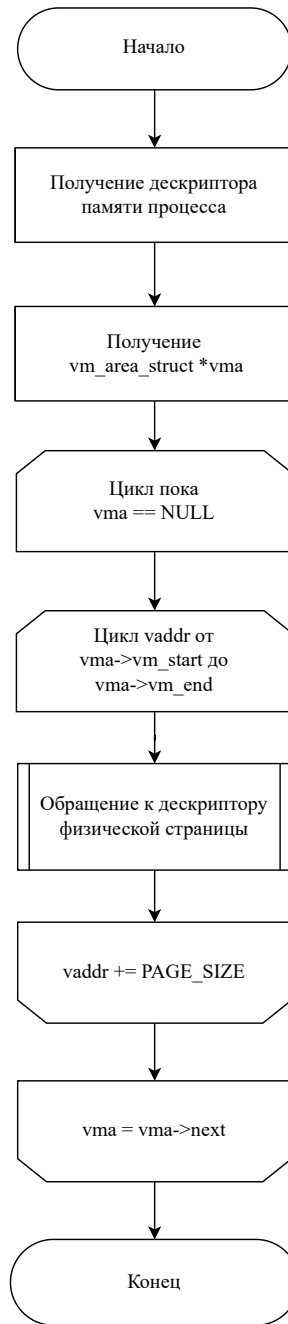


Рисунок 2.3 – Алгоритм сканирования виртуальных страниц

2.3 Алгоритм обращения к дескриптору физической страницы

На рисунке 2.4 показана схема алгоритма обращения к дескриптору физической страницы.

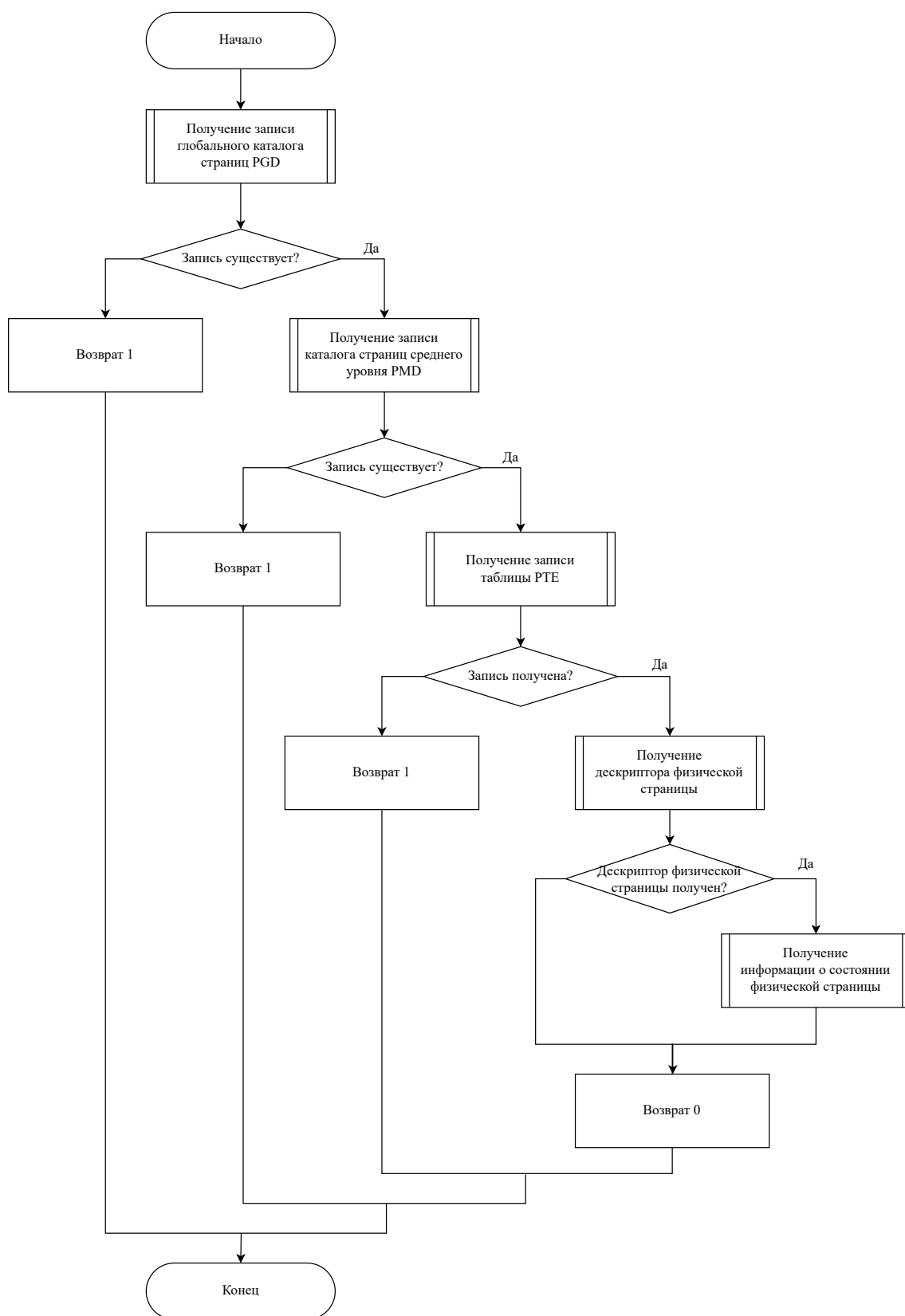


Рисунок 2.4 – Алгоритм обращения к дескриптору физической страницы

2.4 Алгоритм получения информации о состоянии физической страницы

На рисунке 2.5 приведена схема алгоритма получения информации о состоянии физической страницы.



Рисунок 2.5 – Алгоритм получения информации о состоянии физической страницы

2.5 Структура программного обеспечения

На рисунке 2.6 представлена структура программного обеспечения.

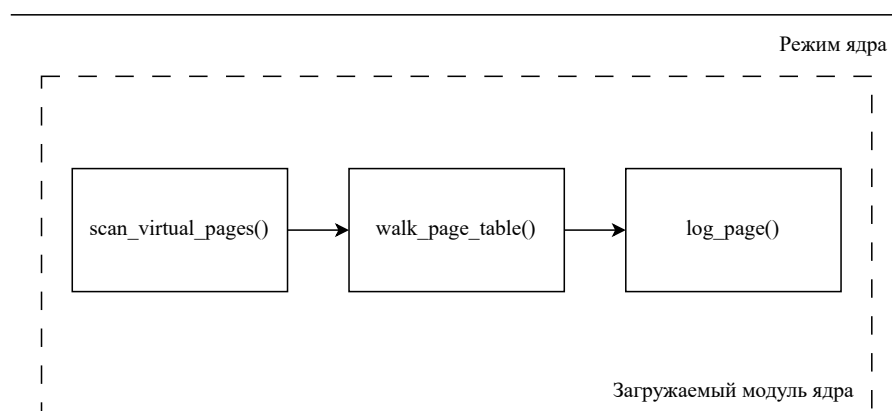


Рисунок 2.6 – Структура программного обеспечения

3 Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования был выбран язык C [4], так как большая часть исходного кода ядра Linux, его модулей и драйверов написана на C.

В качестве среды разработки была выбрана Visual Studio Code [5] в связи с простотой и удобством использования.

3.2 Функции загрузки и выгрузки модуля

При загрузке модуль ядра получает идентификатор процесса, с использованием которого находится дескриптор процесса. Если идентификатор процесса не будет задан, мониторинг не будет произведен, в системный журнал будет выведено соответствующее сообщение.

Функции загрузки и выгрузки модуля показаны в листинге 3.1.

Листинг 3.1 – Функции загрузки и выгрузки модуля

```
1 static int __init md_init(void)
2 {
3     printk(KERN_INFO "pagetracer: module loaded.\n");
4
5     if (process_id != 0)
6     {
7         struct task_struct *task = pid_task(find_vpid(process_id),
8             PIDTYPE_PID);
9         printk(KERN_INFO "pagetracer: %d: process name is %s.\n",
10             process_id, task->comm);
11
12         scan_virtual_pages(task);
13     }
14     else
15     {
16         printk(KERN_ERR "pagetracer: process id was not set.\n");
17     }
18
19     return 0;
20 }
21
22 static void __exit md_exit(void)
23 {
24     printk(KERN_INFO "pagetracer: module unloaded.\n");
25 }
```

3.3 Функция сканирования виртуальных страниц

Алгоритм сканирования виртуальных страниц представлен в листинге 3.2.

Листинг 3.2 – Функция сканирования виртуальных страниц

```
1 static void scan_virtual_pages(struct task_struct *task)
2 {
3     struct mm_struct *mm = task->mm;
4     struct vm_area_struct *vma = mm->mmap;
5
6     unsigned long vaddr;
7     int page_number = 0;
8
9     for (; vma != NULL; vma = vma->vm_next)
10    {
11        for (vaddr = vma->vm_start; vaddr < vma->vm_end; vaddr += PAGE_SIZE)
12        {
13            printk(KERN_INFO "pagetracer: %d: page number is %d.\n",
14                    process_id, page_number++);
15
16            if (walk_page_table(mm, vaddr) != 0)
17            {
18                printk(KERN_INFO "pagetracer: %d: page not mapped in page
19                           table.\n",
20                           process_id);
21            }
22        }
23    }
```


3.4 Функция обращения к дескриптору физической страницы

Алгоритм функции обращения к дескриптору физической страницы приведен в листинге 3.3.

Листинг 3.3 – Функция обращения к дескриптору физической страницы

```
1 static int walk_page_table(struct mm_struct *mm, unsigned long vaddr)
2 {
3     pgd_t *pgd;
4     p4d_t* p4d;
5     pud_t *pud;
6     pmd_t *pmd;
7     pte_t *ptep, pte;
8
9     struct page *page = NULL;
10
11     pgd = pgd_offset(mm, vaddr);
12
13     if (pgd_none(*pgd) || pgd_bad(*pgd))
14         return 1;
15
16     p4d = p4d_offset(pgd, vaddr);
17
18     if (p4d_none(*p4d) || p4d_bad(*p4d))
19         return 1;
20
21     pud = pud_offset(p4d, vaddr);
22
23     if (pud_none(*pud) || pud_bad(*pud))
24         return 1;
25
26     pmd = pmd_offset(pud, vaddr);
27
28     if (pmd_none(*pmd) || pmd_bad(*pmd))
29         return 1;
30
31     ptep = pte_offset_map(pmd, vaddr);
32
33     if (!ptep)
34         return 1;
35
36     pte = *ptep;
37     page = pte_page(pte);
38
39     if (page)
40         log_page(page);
41 }
```

```
42     pte_unmap(pte);
43     return 0;
44 }
```

3.5 Функция получения информации о состоянии физической страницы

Алгоритм функции получения информации о состоянии физической страницы показан в листинге 3.4.

Листинг 3.4 – Функция получения информации о состоянии физической страницы

```
1 static void log_page(struct page *page)
2 {
3     int page_type = (int)page->page_type;
4     printk(KERN_INFO "pagetracer: %d: page type is %d.\n",
5                 process_id, page_type);
6
7     int ref_count = page_ref_count(page);
8     printk(KERN_INFO "pagetracer: %d: page usage count is %d.\n",
9                 process_id, ref_count);
10
11     int map_count = atomic_read(&page->_mapcount);
12     printk(KERN_INFO "pagetracer: %d: references count in page table is
13                 %d.\n",
14                 process_id, map_count);
15 }
```

4 Исследовательский раздел

Программное обеспечение было реализовано на дистрибутиве Ubuntu 20.04 [6], ядро версии 5.15.60 [7].

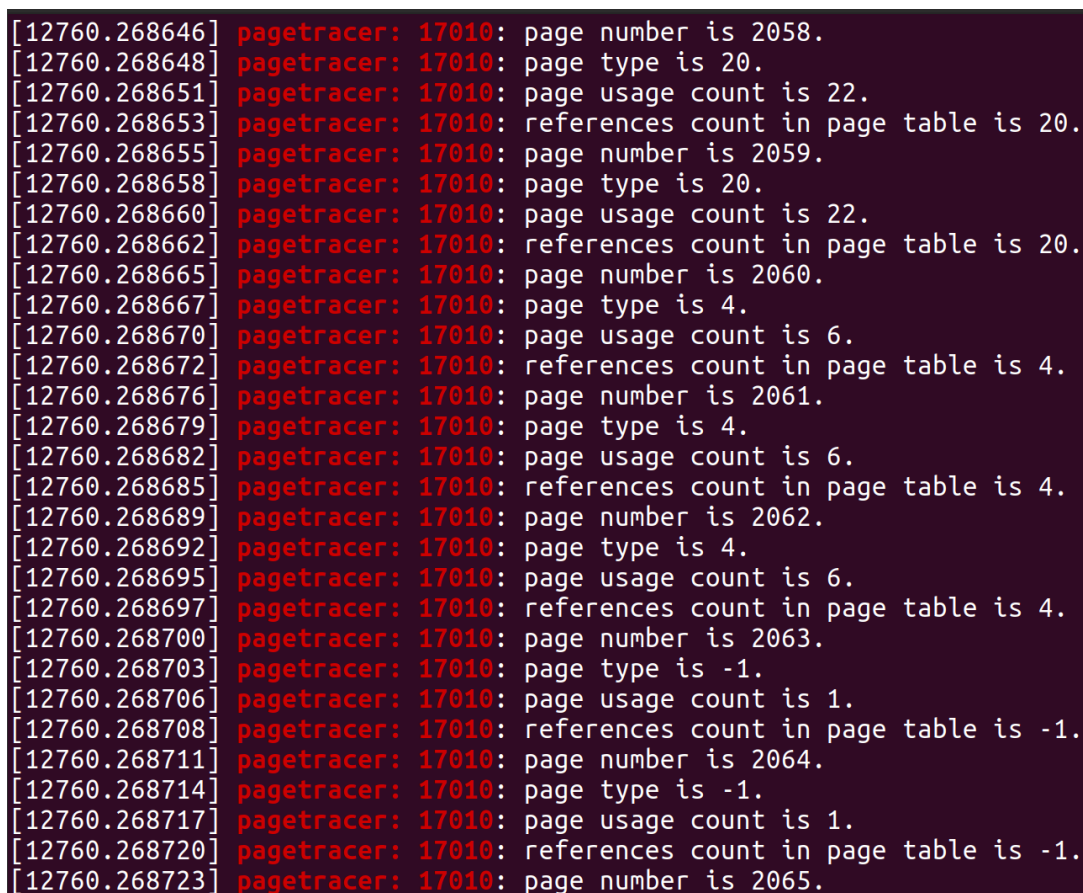
4.1 Примеры работы разработанного ПО

Записи о состоянии физических страниц, выделенных процессу, можно прочитать в системном журнале. Для этого необходимо выполнить команду (листинг 4.1):

Листинг 4.1 – Команда получения записей в системном журнале о состоянии физических страниц, выделенных процессу

```
dmesg | grep "pagetracer: <pid>:"
```

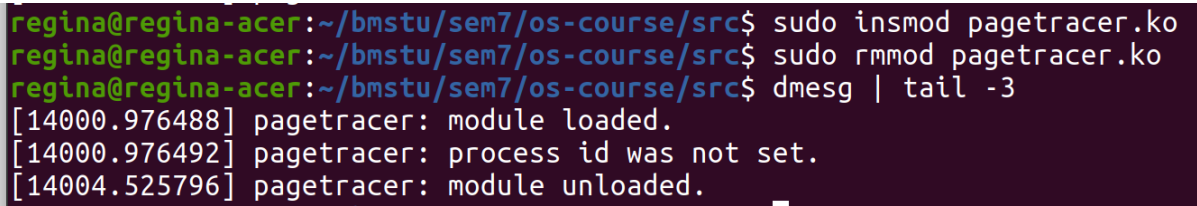
На рисунке 4.1 представлен пример результата работы загружаемого модуля ядра — мониторинг состояния физических страниц процесса с `pid`, равным 17010. Для наглядности представлена часть записей.



```
[12760.268646] pagetracer: 17010: page number is 2058.  
[12760.268648] pagetracer: 17010: page type is 20.  
[12760.268651] pagetracer: 17010: page usage count is 22.  
[12760.268653] pagetracer: 17010: references count in page table is 20.  
[12760.268655] pagetracer: 17010: page number is 2059.  
[12760.268658] pagetracer: 17010: page type is 20.  
[12760.268660] pagetracer: 17010: page usage count is 22.  
[12760.268662] pagetracer: 17010: references count in page table is 20.  
[12760.268665] pagetracer: 17010: page number is 2060.  
[12760.268667] pagetracer: 17010: page type is 4.  
[12760.268670] pagetracer: 17010: page usage count is 6.  
[12760.268672] pagetracer: 17010: references count in page table is 4.  
[12760.268676] pagetracer: 17010: page number is 2061.  
[12760.268679] pagetracer: 17010: page type is 4.  
[12760.268682] pagetracer: 17010: page usage count is 6.  
[12760.268685] pagetracer: 17010: references count in page table is 4.  
[12760.268689] pagetracer: 17010: page number is 2062.  
[12760.268692] pagetracer: 17010: page type is 4.  
[12760.268695] pagetracer: 17010: page usage count is 6.  
[12760.268697] pagetracer: 17010: references count in page table is 4.  
[12760.268700] pagetracer: 17010: page number is 2063.  
[12760.268703] pagetracer: 17010: page type is -1.  
[12760.268706] pagetracer: 17010: page usage count is 1.  
[12760.268708] pagetracer: 17010: references count in page table is -1.  
[12760.268711] pagetracer: 17010: page number is 2064.  
[12760.268714] pagetracer: 17010: page type is -1.  
[12760.268717] pagetracer: 17010: page usage count is 1.  
[12760.268720] pagetracer: 17010: references count in page table is -1.  
[12760.268723] pagetracer: 17010: page number is 2065.
```

Рисунок 4.1 – Мониторинг состояния физических страниц процесса с `pid`, равным 17010

На рисунке 4.2 показана ситуация, когда идентификатор процесса не был задан при загрузке модуля.



```
regina@regina-acer:~/bmstu/sem7/os-course/src$ sudo insmod pagetracer.ko
regina@regina-acer:~/bmstu/sem7/os-course/src$ sudo rmmod pagetracer.ko
regina@regina-acer:~/bmstu/sem7/os-course/src$ dmesg | tail -3
[14000.976488] pagetracer: module loaded.
[14000.976492] pagetracer: process id was not set.
[14004.525796] pagetracer: module unloaded.
```

Рисунок 4.2 – Идентификатор процесса не задан при загрузке модуля

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был разработан загружаемый модуль ядра, позволяющий проводить мониторинг состояния страниц физической памяти, выделенных процессу.

В результате разработки загружаемого модуля ядра удалось при сканировании виртуальных страниц процесса получить доступ к физическим страницам и обратиться к полям дескриптора физической страницы.

В ходе написания работы были решены поставленные задачи:

- проанализированы структуры и функции ядра, предоставляющие информацию о физических страницах, выделенных процессу;
- разработаны алгоритмы и структура программного обеспечения;
- реализовано программное обеспечение;
- проведен анализ работы разработанного программного обеспечения.

Цель работы достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лав Р. Ядро Linux: описание процесса разработки, 3-е изд. — М.: ООО «И.Д. Вильямс», 2013. — с. 496.
2. struct page, the Linux physical page frame data structure [Электронный ресурс]. — Режим доступа: <https://blogs.oracle.com/linux/post/struct-page-the-linux-physical-page-frame-data-structure> (дата обращения: 25.12.2022).
3. Page Table Management [Электронный ресурс]. — Режим доступа: <https://www.kernel.org/doc/gorman/html/understand/understand006.html> (дата обращения: 25.12.2022).
4. ISO/IEC 9899:1990 Programming languages — C [Электронный ресурс]. — Режим доступа: <https://www.iso.org/standard/17782.html> (дата обращения: 25.12.2022).
5. Visual Studio Code [Электронный ресурс]. — Режим доступа: <https://code.visualstudio.com/> (дата обращения: 25.12.2022).
6. Ubuntu 20.04.5 LTS (Focal Fossa) [Электронный ресурс]. — Режим доступа: <https://releases.ubuntu.com/focal/> (дата обращения: 25.12.2022).
7. Исходный код ядра Linux версии 5.15.60 [Электронный ресурс]. — Режим доступа: <https://elixir.bootlin.com/linux/v5.15.60/source> (дата обращения: 25.12.2022).

ПРИЛОЖЕНИЕ А

Исходный код загружаемого модуля

Листинг А.1 – Загружаемый модуль ядра

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/moduleparam.h>
5  #include <linux/sched.h>
6  #include <linux/mm.h>
7  #include <linux/pid.h>
8  #include <linux/page_ref.h>
9  #include <linux/page-flags.h>
10 #include <linux/atomic.h>
11 #include <asm/page.h>
12
13
14 MODULE_LICENSE("GPL");
15 MODULE_AUTHOR("Khamzina Regina");
16 MODULE_DESCRIPTION("Module for monitoring the state of pages allocated to a
    process");
17
18
19 static int process_id;
20 module_param(process_id, int, 0);
21
22
23 static void log_page(struct page *page)
24 {
25     int page_type = (int)page->page_type;
26     printk(KERN_INFO "pagetracer: %d: page type is %d.\n",
27             process_id, page_type);
28
29     int ref_count = page_ref_count(page);
30     printk(KERN_INFO "pagetracer: %d: page usage count is %d.\n",
31             process_id, ref_count);
32
33     int map_count = atomic_read(&page->_mapcount);
34     printk(KERN_INFO "pagetracer: %d: references count in page table is
        %d.\n",
35             process_id, map_count);
36 }
37
38 static int walk_page_table(struct mm_struct *mm, unsigned long vaddr)
39 {
40     pgd_t *pgd;
41     p4d_t* p4d;
```

```

42     pud_t *pud;
43     pmd_t *pmd;
44     pte_t *ptep, pte;
45
46     struct page *page = NULL;
47
48     pgd = pgd_offset(mm, vaddr);
49
50     if (pgd_none(*pgd) || pgd_bad(*pgd))
51         return 1;
52
53     p4d = p4d_offset(pgd, vaddr);
54
55     if (p4d_none(*p4d) || p4d_bad(*p4d))
56         return 1;
57
58     pud = pud_offset(p4d, vaddr);
59
60     if (pud_none(*pud) || pud_bad(*pud))
61         return 1;
62
63     pmd = pmd_offset(pud, vaddr);
64
65     if (pmd_none(*pmd) || pmd_bad(*pmd))
66         return 1;
67
68     ptep = pte_offset_map(pmd, vaddr);
69
70     if (!ptep)
71         return 1;
72
73     pte = *ptep;
74     page = pte_page(pte);
75
76     if (page)
77         log_page(page);
78
79     pte_unmap(ptep);
80     return 0;
81 }
82
83 static void scan_virtual_pages(struct task_struct *task)
84 {
85     struct mm_struct *mm = task->mm;
86     struct vm_area_struct *vma = mm->mmap;
87
88     unsigned long vaddr;
89     int page_number = 0;

```



```

90
91     for (; vma != NULL; vma = vma->vm_next)
92     {
93         for (vaddr = vma->vm_start; vaddr < vma->vm_end; vaddr += PAGE_SIZE)
94         {
95             printk(KERN_INFO "pagetracer: %d: page number is %d.\n",
96                     process_id, page_number++);
97
98             if (walk_page_table(mm, vaddr) != 0)
99             {
100                 printk(KERN_INFO "pagetracer: %d: page not mapped in page
101                             table.\n",
102                             process_id);
103             }
104         }
105     }
106
107 static int __init md_init(void)
108 {
109     printk(KERN_INFO "pagetracer: module loaded.\n");
110
111     if (process_id != 0)
112     {
113         struct task_struct *task = pid_task(find_vpid(process_id),
114             PIDTYPE_PID);
115         printk(KERN_INFO "pagetracer: %d: process name is %s.\n",
116             process_id, task->comm);
117
118         scan_virtual_pages(task);
119     }
120     else
121     {
122         printk(KERN_ERR "pagetracer: process id was not set.\n");
123     }
124
125     return 0;
126 }
127
128 static void __exit md_exit(void)
129 {
130     printk(KERN_INFO "pagetracer: module unloaded.\n");
131 }
132
133 module_init(md_init);
134 module_exit(md_exit);

```

Листинг А.2 – <asm/page.h>

```

1  #ifndef __ASM_ARC_PAGE_H
2  #define __ASM_ARC_PAGE_H
3
4  #include <uapi/asm/page.h>
5
6  #ifdef CONFIG_ARC_HAS_PAE40
7
8  #define MAX_POSSIBLE_PHYSMEM_BITS    40
9  #define PAGE_MASK_PHYS              (0xff00000000ull | PAGE_MASK)
10
11 #else /* CONFIG_ARC_HAS_PAE40 */
12
13 #define MAX_POSSIBLE_PHYSMEM_BITS    32
14 #define PAGE_MASK_PHYS              PAGE_MASK
15
16 #endif /* CONFIG_ARC_HAS_PAE40 */
17
18 #ifndef __ASSEMBLY__
19
20 #define clear_page(paddr)            memset((paddr), 0, PAGE_SIZE)
21 #define copy_user_page(to, from, vaddr, pg) copy_page(to, from)
22 #define copy_page(to, from)         memcpy((to), (from), PAGE_SIZE)
23
24 struct vm_area_struct;
25 struct page;
26
27 #define __HAVE_ARCH_COPY_USER_HIGHPAGE
28
29 void copy_user_highpage(struct page *to, struct page *from,
30                        unsigned long u_vaddr, struct vm_area_struct *vma);
31 void clear_user_page(void *to, unsigned long u_vaddr, struct page *page);
32
33 typedef struct {
34     unsigned long pgd;
35 } pgd_t;
36
37 #define pgd_val(x)    ((x).pgd)
38 #define __pgd(x)      ((pgd_t) { (x) })
39
40 #if CONFIG_PGTABLE_LEVELS > 3
41
42 typedef struct {
43     unsigned long pud;
44 } pud_t;
45
46 #define pud_val(x)    ((x).pud)
47 #define __pud(x)      ((pud_t) { (x) })

```

```

48
49 #endif
50
51 #if CONFIG_PGTABLE_LEVELS > 2
52
53 typedef struct {
54     unsigned long pmd;
55 } pmd_t;
56
57 #define pmd_val(x) ((x).pmd)
58 #define __pmd(x) ((pmd_t) { (x) })
59
60 #endif
61
62 typedef struct {
63 #ifdef CONFIG_ARC_HAS_PAE40
64     unsigned long long pte;
65 #else
66     unsigned long pte;
67 #endif
68 } pte_t;
69
70 #define pte_val(x) ((x).pte)
71 #define __pte(x) ((pte_t) { (x) })
72
73 typedef struct {
74     unsigned long pgprot;
75 } pgprot_t;
76
77 #define pgprot_val(x) ((x).pgprot)
78 #define __pgprot(x) ((pgprot_t) { (x) })
79 #define pte_pgprot(x) __pgprot(pte_val(x))
80
81 typedef struct page *pgtable_t;
82
83 /*
84  * Use virt_to_pfn with caution:
85  * If used in pte or paddr related macros, it could cause truncation
86  * in PAE40 builds
87  * As a rule of thumb, only use it in helpers starting with virt_
88  * You have been warned !
89  */
90 #define virt_to_pfn(kaddr) (__pa(kaddr) >> PAGE_SHIFT)
91
92 /*
93  * When HIGHMEM is enabled we have holes in the memory map so we need
94  * pfn_valid() that takes into account the actual extents of the physical
95  * memory

```

```

96  */
97  #ifdef CONFIG_HIGHMEM
98
99  extern unsigned long arch_pfn_offset;
100 #define ARCH_PFN_OFFSET      arch_pfn_offset
101
102 extern int pfn_valid(unsigned long pfn);
103 #define pfn_valid            pfn_valid
104
105 #else /* CONFIG_HIGHMEM */
106
107 #define ARCH_PFN_OFFSET      virt_to_pfn(CONFIG_LINUX_RAM_BASE)
108 #define pfn_valid(pfn)      (((pfn) - ARCH_PFN_OFFSET) < max_mapnr)
109
110 #endif /* CONFIG_HIGHMEM */
111
112 /*
113  * __pa, __va, virt_to_page (ALERT: deprecated, don't use them)
114  *
115  * These macros have historically been misnamed
116  * virt here means link-address/program-address as embedded in object code.
117  * And for ARC, link-addr = physical address
118  */
119 #define __pa(vaddr)          ((unsigned long)(vaddr))
120 #define __va(paddr)          ((void *)((unsigned long)(paddr)))
121
122 #define virt_to_page(kaddr)  pfn_to_page(virt_to_pfn(kaddr))
123 #define virt_addr_valid(kaddr)  pfn_valid(virt_to_pfn(kaddr))
124
125 /* Default Permissions for stack/heaps pages (Non Executable) */
126 #define VM_DATA_DEFAULT_FLAGS  VM_DATA_FLAGS_NON_EXEC
127
128 #define WANT_PAGE_VIRTUAL    1
129
130 #include <asm-generic/memory_model.h>    /* page_to_pfn, pfn_to_page */
131 #include <asm-generic/getorder.h>
132
133 #endif /* !__ASSEMBLY__ */
134
135 #endif

```