СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

В многопроцессорных системах, основным ресурсом является физическая память. Процесс не может начать или продолжить свое выполнение, если ему не выделена запрошенная страница, при выделении которой возникает страничное прерывание. Частота страничных прерываний, возникающих у конкретного процесса является показателем эффективности его выполнения. В связи с этим, необходимо иметь возможность отслеживать страничные прерывания процессов.

Таким образом, было принято решение о реализации ПО, позволяющего производить мониторинг выделения памяти процессам.

# 1. Аналитический раздел

## 1.1. Постановка задачи

В соответствии с заданием к курсовой работе, необходимо разработать загружаемый модуль ядра Linux, осуществляющий мониторинг страничных прерываний процессов. Для решения поставленной задачи необходимо:

1) проанализировать структуры ядра, отвечающие за хранение информации о выделенной процессу памяти;

2) проанализировать системные вызовы и порядок их выполнения при возникновении страничных прерываний;

3) реализовать загружаемый модуль ядра, позволяющий собирать информацию о выделении памяти процессам.

К загружаемому модулю ядра были выдвинуты следующие требования:

1) модуль должен обнаруживать страничные прерывания;

2) при возникновении страничного прерывания, модуль должен определять, какой процесс его вызвал и сохранять информацию о нем;

3) модуль должен сохранять время первого и последнего обновления информации о процессе;

4) модуль должен иметь возможность осуществлять передачу данных из пространства ядра в пространство пользователя.

## 1.2. Структуры ядра, хранящие информацию о выделенной процессу памяти

В решаемой задаче анализируются процессы, размер физических страниц для которых равен 4096 байт, или 4 КБ [1]. Это значение можно

получить при обращении к константе *PAGE_SIZE*. Виртуальная память, выделяемая процессу, разделена на области VMA - virtual memory area - описывающие участки адресного пространства процесса [1].

Процессы в Linux представлены структурой *task_struct*. Одно из полей данной структуры - *struct mm_struct *mm* - является указателем на структуру *mm_struct*, которая содержит список VMA, таблицы страниц и счетчики этих страниц [2]. Часть описания данной структуры представлена в листинге 1, где [4]:

1) vm_area_struct *mmap - указатель на голову списка областей VMA;

2) struct mm_rss_stat rss_stat - структура, описывающая счетчики физических страниц, выделенных процессу.

```
1 struct mm_struct {
2   struct vm_area_struct *mmap;
3   ...
4   struct mm_rss_stat rss_stat;
5   ...
6 }
```

Листинг 1. Часть структуры *mm_struct*

В листинге 2 приведена часть структуры *vm_area_struct*. Эта структура предназначена для хранения информации о виртуальной памяти, выделенной процессу.

```
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 }
```

Листинг 2. Часть структуры *vm_area_struct*

В данной структуре, *vm_start* - адрес начала области виртуальной памяти, описываемой данной структурой, *vm_end* - следующий бит после последнего адреса этой области. Для каждого процесса может быть

выделено несколько VMA, которые представлены в памяти в виде двусвязного списка, где $*vm\_next$ и $*vm\_prev$ - указатели на следующую и предыдущую области виртуальной памяти, соответственно [4].

В листинге 3 приведена структура *mm_rss_stat*, в которой поле $count[NR\_MM\_COUNTERS]$ хранит счетчики выделенных процессу физических страниц.

```
1 struct mm_rss_stat {
2   atomic_long_t count[NR_MM_COUNTERS];
3 };
```

Листинг 3. Структура *mm_rss_stat*

Получить информацию, хранящуюся в данной структуре можно с помощью функции $get\_mm\_rss(struct\ mm\_struct\ *mm)$, представленной в листинге 4 [4].

```
1 static inline unsigned long get_mm_rss(struct mm_struct *mm)
2 {
3   return get_mm_counter(mm, MM_FILEPAGES) +
4     get_mm_counter(mm, MM_ANONPAGES) +
5     get_mm_counter(mm, MM_SHMEMPAGES);
6 }
```

Листинг 4. Функция $get\_mm\_rss(struct\ mm\_struct\ *mm)$

Данная функция возвращает суммарное число страниц, выделенных процессу, где [4]:

1) *MM_FILEPAGES* - страницы, выделенные для файлов процесса;

2) *MM_ANONPAGES* - анонимные страницы;

3) *MM_SHMEMPAGES* - разделяемые страницы.

Помимо информации о выделенной физической и виртуальной памяти, структура *task_struct* хранит в себе информацию об идентификаторе процесса, а также о его имени.

## 1.3. Страничные прерывания

При выполнении, процесс запрашивает физические страницы. Если запрашиваемая страница отсутствует в оперативной памяти, то возникает страничное прерывание [3]. Страничное прерывание может означать как ошибку, так и запрос на выделение памяти, что и следует определить обработчику страничного прерывания.

В общем виде, схема, демонстрирующая процесс определения причины страничного прерывания представлена на рисунке 1.

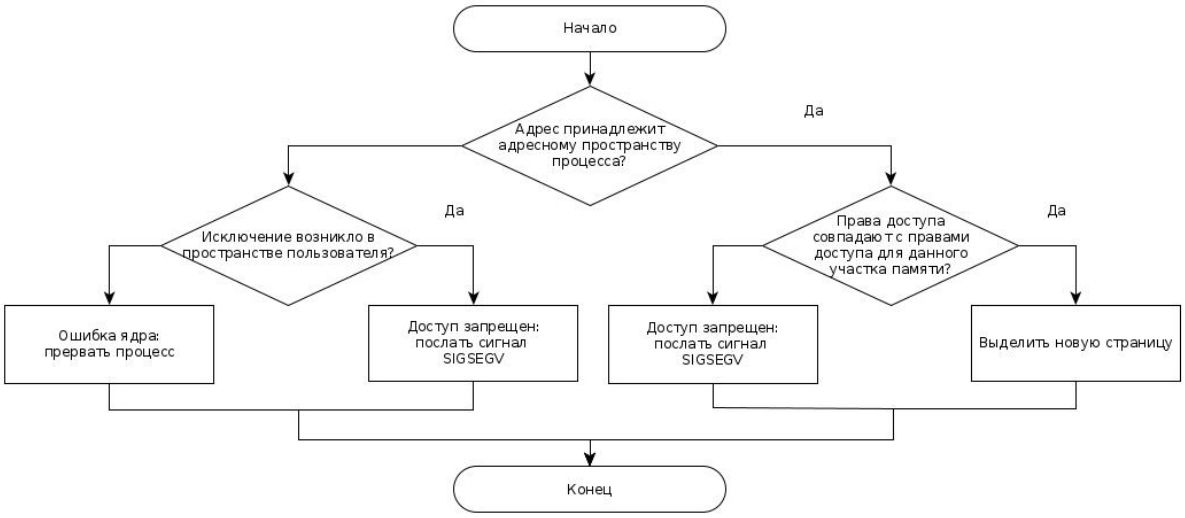



Рисунок 1. Схема определения причины страничного прерывания

На рисунке 2 представлены этапы обслуживания страничного прерывания.

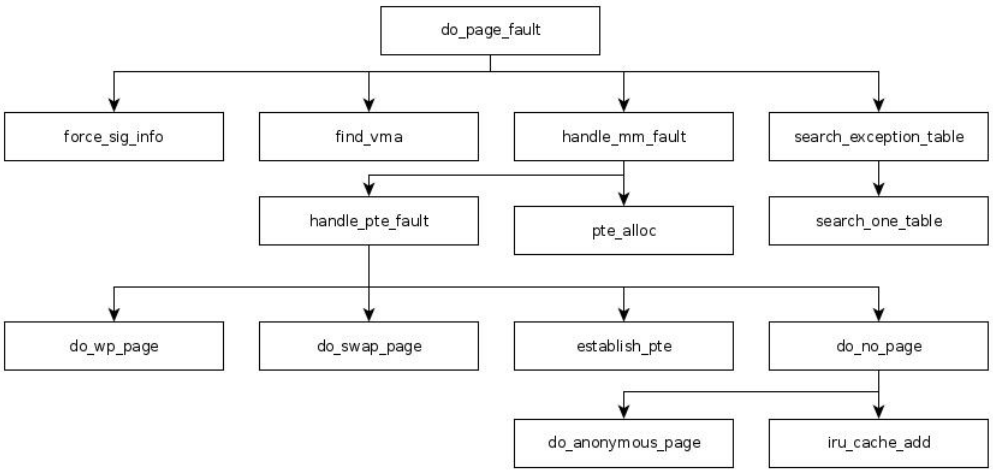



Рисунок 2. Этапы обслуживания страничных прерываний

Из данной схемы и из исходного кода вызова *do_page_fault*() [4], представленного в листинге 5, видно, что основным обработчиком страничного прерывания является обработчик *handle_mm_fault*, внутри которого выделяется память и выполняется проверка, возникли ли при этом ошибки.

```
asmlinkage void do_page_fault(unsigned long address, unsigned long mmcsr,
    long cause, struct pt_regs *regs)
{
  ...
  fault = handle_mm_fault(vma, address, flags);
  ...
  if (unlikely(fault & VM_FAULT_ERROR)) {
    if (fault & VM_FAULT_OOM)
      goto out_of_memory;
    else if (fault & VM_FAULT_SIGSEGV)
      goto bad_area;
    else if (fault & VM_FAULT_SIGBUS)
      goto do_sigbus;
    BUG();
  }
}
```

Листинг 5. Системный вызов *do_page_fault*

Таким образом, для того чтобы отследить возникновение страничных прерываний, необходимо обнаружить вызов *handle_mm_fault*.

## 1.4. Обнаружение страничного прерывания

Все системные вызовы проходят через таблицу *sys_call_table*. Индекс обработчика соответствует номеру системного вызова. Для того, чтобы отследить страничное прерывание, можно подменить существующий системный вызов в данной таблице своим [6]. Данный метод сложен и неудобен в использовании, ведь приходится переписывать обработчики системных вызовов, что, при некорректном использовании, может повредить системе. Также, данный метод приводит к необходимости хранить замененный обработчик, чтобы иметь возможность его восстановить при выгрузке модуля.

Таким образом, желательно, найти способ обнаружения страничного прерывания методом, не предполагающим внесения изменений в ядро системы.

***Kprobe*** - механизм отладки для ядра Linux, также используемый для отслеживания событий внутри системы. Их можно использовать для мониторинга различных функций, журналирования событий и отслеживания ошибок [5]. В дальнейшем, функции, системные вызовы, события и ошибки, для которых может быть поставлен ***kprobe***, будем именовать инструкцией.

Использование ***kprobe*** заключается в установке обработчиков на определенную инструкцию. Чтобы поставить ***kprobe***, необходимо задать адрес инструкции, которую мы хотим отслеживать, а также определить предобработчик и постобработчик. Предобработчик вызывается до выполнения инструкции, на которую поставлен ***kprobe***, а постобработчик вызывается сразу по завершении выполнения данной инструкции.

***Kprobe*** описывается структурой, представленной в листинге 6, где [4]:

1) $*addr$ - адрес установки ***kprobe***;
2) $pre\_handler$ - функция, вызываемая до инструкции;
3) $post\_handler$ - функция, вызываемая после инструкции;
4) $fault\_handler$ - функция, вызываемая в случае возникновении ошибки, при выполнении инструкции.

```
1 struct kprobe {
2   ...
3   kprobe_opcode_t *addr;
4   ...
5   kprobe_pre_handler_t pre_handler;
6   kprobe_post_handler_t post_handler;
7   kprobe_fault_handler_t fault_handler;
8   ...
9 };
```

Листинг 6. Структура ***kprobe***

Таким образом, ***kprobe*** является простым и удобным механизмом отслеживания возникновения событий в системе.

# 2. Конструкторский раздел

## 2.1. Разработка структуры для хранения информации о процессе

Для мониторинга выделения памяти процессам необходимо хранить следующую информацию:

1) объем выделенной виртуальной памяти;
2) объем выделенной физической памяти;
3) количество страничных прерываний, зафиксированных загружаемым модулем ядра;
4) идентификатор процесса;
5) его имя;
6) дату и время первого и последнего обновления информации.

При работе в режиме ядра можно получить информацию о процессе, вызвавшем ту или иную функцию. В файле *linux/shed.h* хранится указатель $task\_struct * current$, указывающий на текущий процесс [4]. Таким образом, при возникновении страничного прерывания, можно получить информацию о процессе, вызвавшем его.

При повторной обработке страничного прерывания процесса, необходимо обновлять полученную ранее информацию. Для большого числа процессов для этих целей удобно завести связный список структур. При вызове обработчика, необходимо будет проверить, обрабатывался ли до этого данный процесс, и если нет - создать новый элемент списка. Разработанная структура должна содержать следующие поля:

1) pid - идентификатор процесса;
2) comm - имя процесса;

3) vma - объем виртуальной памяти, выделенной процессу, в килобайтах;

4) rss - объем физической памяти, выделенной процессу, в килобайтах;

5) page_fault_counter - счетчик страничных прерываний, зафиксированных модулем;

6) first_noticed - время первой обработки процесса модулем;

7) last_updated - время последней обработки процесса модулем;

8) next - указатель на следующий элемент списка.

## 2.2. Разработка алгоритма обработки страничного прерывания

На рисунке 3 представлен алгоритм обработки страничного прерывания внутри загружаемого модуля ядра.

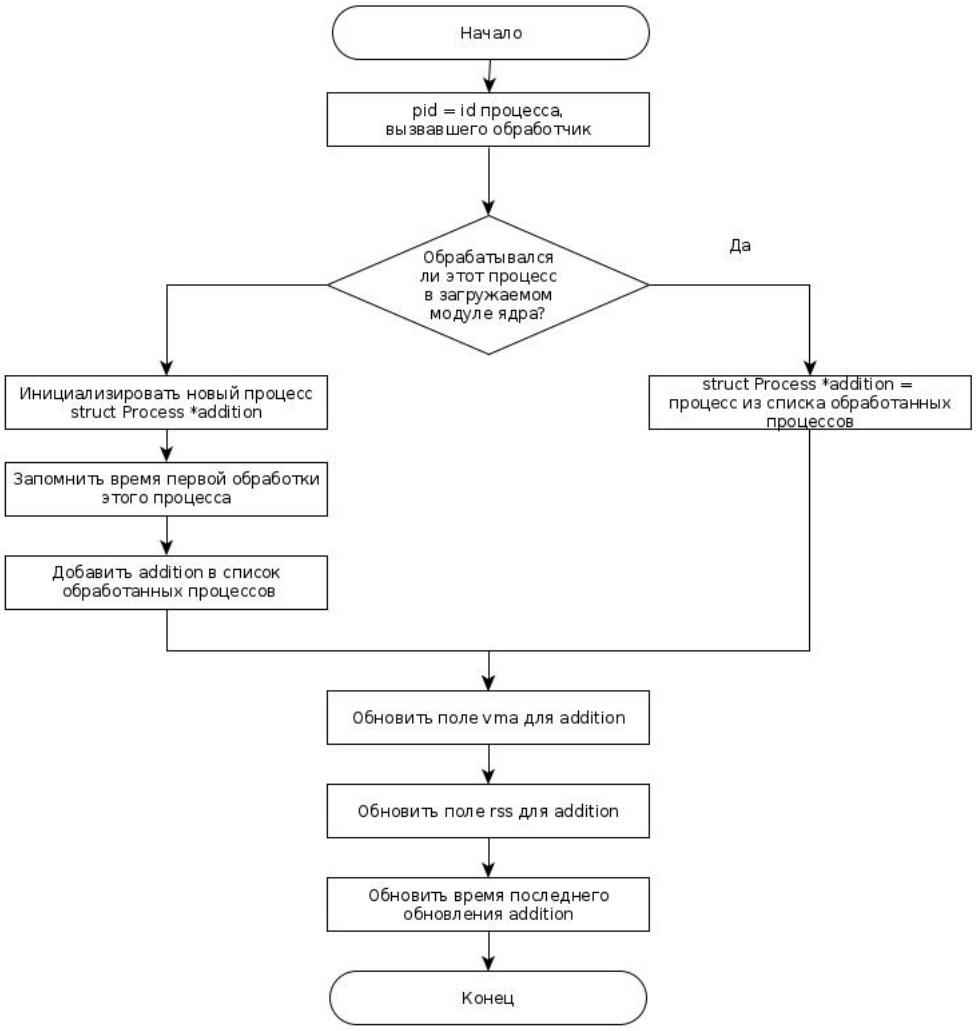



Рисунок 3. Схема алгоритма обработки страничного прерывания

## 2.3. Разработка схемы передачи информации в режим пользователя

Как было показано в пункте 2.1, данные, полученные во время мониторинга записываются в связный список структур. Для вывода информации, полученной в результате мониторинга в режиме пользователя необходимо перевести информацию, содержащуюся в списке структур, в текстовое предсталение, для чего следует реализовать функцию *create_buffer*. Данная функция должна принимать на вход список, и составлять из него текстовое сообщение, которое затем необходимо передать в пространство пользователя. Для этого используется специальная функция ядра *copy_to_user*, предназначенная для передачи данных из

пространства ядра в пространство пользователя.

Саму операцию передачи данных можно осуществлять через виртуальную файловую систему */proc*, где для созданного файла, с помощью структуры *file_operations*, определить функцию чтения. Схема получения данных из загружаемого модуля ядра представлена на рисунке 4.

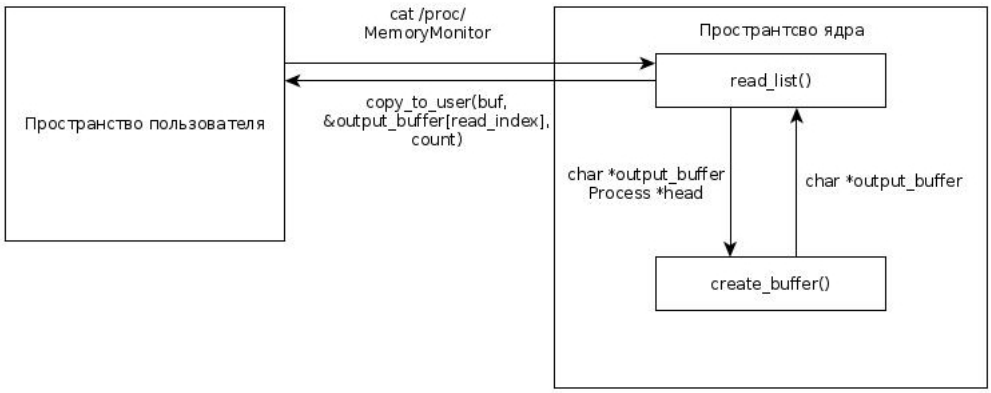



Рисунок 4. Схема передачи данных мониторинга из пространства ядра в пространство пользователя

# 3. Технологический раздел

## 3.1. Выбор среды и языка программирования

Для реализации загружаемого модуля ядра использовался язык программирования C. Для сборки был использован Makefile, упрощающий процесс компиляции и сборки исполняемого файла. Были задействованы библиотеки, дающие возможность использовать структуры ядра Linux.

## 3.2. Требования к аппаратуре

Были выдвинуты следующие требования к аппаратуре:

1) операционная система семейста Linux;
2) версия ядра 4.15.

## 3.3. Загружаемый модуль ядра

### 3.3.1. Инициализация модуля

В функции ***myinit***() выполняются следующие действия.

1. Выделение памяти буферу timeBuf и output_buffer, а также инициализация головы списка процессов head. Буфер timeBuf предназначен для временного хранения даты и времени обновления информации о процессе перед занесением в структуру Process. output_buffer предназначен для записи сообщения, передаваемого пространству пользователя.
2. Инициализация структуры file_operations fops и создание файла MemoryMonitor в виртуальной файловой системе */**proc***. На этом

этапе производится инициализация функции чтения, описанной в разделе 2.3. Создание файла выполняется с помощью функции proc_create_data.

3. Инициализация kprobe kp. Инициализируются функции обработчиков, а также адрес обработчика handle_mm_fault, вызов которого и будет отслеживаться. В данной ситуации нас интересует только постобработчик, потому что его вызов следует за завершением обработчика страничного прерывания. Таким образом, если память была выделена процессу, ее уже можно будет посчитать. Регистрация kprobe в системе выполняется функцией register_kprobe().

   В листинге 7 приведен код инициализации модуля ядра.

```
1  int myinit(void)
2  {
3      timeBuf = vmalloc(BYTES);
4      head = NULL;
5
6      output_buffer = vmalloc(4*PAGE_SIZE);
7
8      fops.owner = THIS_MODULE;
9      fops.read = read_list;
10
11     proc_entry = proc_create_data("MemoryMonitor",0666, NULL, &fops, NULL);
12     if (!proc_entry)
13     {
14         vfree(timeBuf);
15         printk(KERN_INFO "MemoryMonitor: Cannot create fortune file.\n");
16         return -ENOMEM;
17     }
18
19     kp.pre_handler = pre_handler;
20     kp.post_handler = post_handler;
21     kp.fault_handler = handler_fault;
22     kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("__handle_mm_fault");
23     register_kprobe(&kp);
24
25     printk(KERN_INFO "MemoryMonitor: kprobe registered.\n");
26     return 0;
27 }
```

Листинг 7. Инициализация загружаемого модуля ядра.

### 3.3.2. Обработчик страничного прерывания

Как было описано в пункте 1.3, основную работу при возникновении страничного прерывания выполняет обработчик *handle_mm_fault*(). С помощью *kprobe*, был написан постобработчик для данного вызова, позволяющий получить информацию о процессе, вызвавшем страничное прерывание, после выделения ему запрошенной памяти. В листинге 8 приведен код постобработчика *post_handler*.

```
1  void post_handler(struct kprobe *p, struct pt_regs *regs, unsigned long
       flags)
2  {
3      struct Process *addition = find_process_by_id(head, current->pid);
4      timeBuf = get_current_time(timeBuf);
5      if (addition == NULL)
6      {
7          ...
8          addition->vma = countVMA(current);
9          push_process(&head, addition);
10     }
11     if (current->mm != NULL)
12     {
13         addition->rss = countRSS(current);
14     }
15     addition->page_fault_counter++;
16     sprintf(addition->last_updated, "%s", timeBuf);
17 }
```

Листинг 8. Постобработчик страничного прерывания.

Функция find_process_by_id() производит поиск процесса в списке обработанных процессов по его идентификатору. Если данного процесса нет в списке, то возвращается NULL, иначе - указатель на элемент списка. В случае, когда был возвращен NULL, создается новый элемент списка, где все поля инициализируются в 0, запоминается pid и comm процесса, а в поле first_noticed записывается текущее время. Также на этом этапе производится вычисление объема виртуальной памяти, выделенной данному процессу, с помощью функции countVMA, реализация которой представлена в листинге 9. Из приведенной в пунтке 1.2 структуры *vm_area_struct*, известно, что объем виртуальной памяти определяется как разница между начальным и конечным адресами области

виртуальной памяти. Полученные таким образом значения для каждого участка VMA суммируются. После этого новый элемент добавляется в список функцией push_process().

```
long countVMA(struct task_struct *task)
{
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    long total = 0;
    mm = task->mm;
    for (vma = mm->mmap ; vma ; vma = vma->vm_next)
    {
        total += vma->vm_end - vma->vm_start;
    }
    return total/BYTES;
}
```

Листинг 9. Реализация функции countVMA.

Функция get_current_time получает текущие дату и время (с помощью системной функции do_gettimeofday) и преобразовывает их в строку формата $dd/mm/yyyy\ hh:mm:ss$.

После того, как элемент списка был найден, для процесса, вызвавшего страничное прерывание подсчитывается объем физической памяти с помощью функций countRSS. Реализации этой функции приведена в листинге 10.

```
long countRSS(struct task_struct *task)
{
    return get_mm_rss(task->mm)*PAGE_SIZE/BYTES;
}
```

Листинг 10. Реализация функции countRSS.

Количество выделенных физических страниц можно узнать с помощью функции *get_mm_rss*(). Размер одной страницы хранится в константе *PAGE_SIZE* в байтах, то есть, чтобы узнать объем выделенной физической памяти в килобайтах, необходимо умножить число страниц на *PAGE_SIZE* и поделить на 1024 байта.

После обновления объемов выделенной памяти, увеличивается счетчик страничных прерываний и фиксируется момент последнего обновления.

### 3.3.3. Реализация связного списка информации о процессах

Для хранения определенной информации о процессах была разработана специальная структура Process, описанная в пункте 2.1. Реализация данной структуры приведена в листинге 11.

```
1  struct Process
2  {
3      long pid;
4      char* comm;
5      long vma;
6      long rss;
7      long page_fault_counter;
8      char* first_noticed;
9      char* last_updated;
10     struct Process *next;
11 };
```

Листинг 11. Структура ***Process***, предназначенная для хранения информации о процессе

Поскольку такая информация должна храниться о всех процессах, запущенных в системе с момента начала работы модуля, структуры организуются в связный список типа FIFO. Для работы со связным списком разработаны следующие функции.

1. find_process_by_id - данная функция предназначена для поиска элемента с идентификатором pid в списке. Для этого, до тех пор, пока не встречен элемент с таким идентификатором, происходит итерация по списку. Реализация данной функции приведена в листинге 12.

```
struct Process *find_process_by_id(struct Process *head, long pid)
{
    struct Process *tmp = NULL;
    struct Process *head_save = head;
    while (head && tmp == NULL)
    {
        if (head->pid == pid)
            tmp = head;
        head = head->next;
    }
    head = head_save;
    return tmp;
}
```

Листинг 12. Реализация функции *find_process_by_id*.

2. push_process - функция предназначена для добавления нового элемента в список. Реализация функции приведена в листинге 13.

```
void push_process(struct Process **head, struct Process *new_proc)
{
    if (*head == NULL)
    {
        *head = new_proc;
        (*head)->next = NULL;
    }
    else
    {
        new_proc->next = *head;
        *head = new_proc;
    }
}
```

Листинг 13. Реализация функции *push_process*.

3. free_list - эта функция вызывается при выгрузке загружаемого модуля для освобождения памяти, выделенной под элементы списка. Код этой функции приведен в листинге 14.

```
void free_list(struct Process *head)
{
    struct Process *tmp = NULL;
    while(head)
    {
        tmp = head->next;
        vfree(head->comm);
        vfree(head);
        head = tmp;
    }
}
```

Листинг 14. Реализация функции *free_list*.

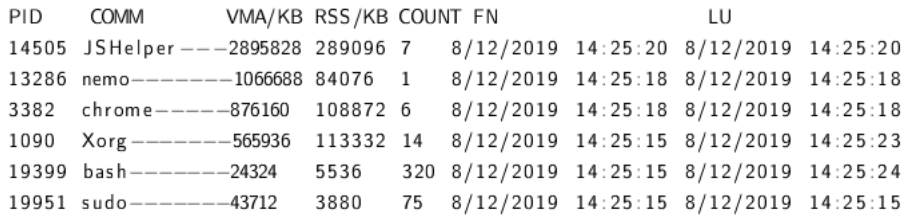
4. create_buffer - данная фукнция была описана в разделе 2.3, она вызывается при обращении к файлу в виртуальной файловой системе proc для того, чтобы по имеющемуся списку собрать текстовое сообщение, предназначенное для отправки в пространство пользователя. Код данной фукнции приведен в листинге 15.

```
char* create_buffer(char * output_buffer, struct Process *head)
{
    char *tmp = vmalloc(BYTES), *tabs;
    memset(tmp, BYTES, 0);
    memset(output_buffer, 4*PAGE_SIZE, 0);
    sprintf(output_buffer, "PID COMM VMA/KB RSS/KB COUN FN LU\n");
    while(head)
    {
        tabs = create_separator(head->comm);
        sprintf(tmp, "%ld\t%s%s%ld\t%ld\t%ld\t%s\t%s\n", head->pid,
                                                head->comm,
                                                tabs,
                                                head->vma,
                                                head->rss,
                                                head->page_fault_counter,
                                                head->first_noticed,
                                                head->last_updated);
        strcat(output_buffer, tmp);
        vfree(tabs);
        head = head->next;
    }
    write_index = strlen(output_buffer);
    return output_buffer;
}
```

Листинг 15. Реализация функции *create_buffer*.

## 3.4. Демонстрация работы загружаемого модуля ядра

Скриншот 5 демонстрирует пример данных, которые были получены во время работы модуля.



| PID | COMM | VMA/KB | RSS/KB | COUNT | FN | LU |
|---|---|---|---|---|---|---|
| 14505 | JSHelper | 2895828 | 289096 | 7 | 8/12/2019 14:25:20 | 8/12/2019 14:25:20 |
| 13286 | nemo | 1066688 | 84076 | 1 | 8/12/2019 14:25:18 | 8/12/2019 14:25:18 |
| 3382 | chrome | 876160 | 108872 | 6 | 8/12/2019 14:25:18 | 8/12/2019 14:25:18 |
| 1090 | Xorg | 565936 | 113332 | 14 | 8/12/2019 14:25:15 | 8/12/2019 14:25:23 |
| 19399 | bash | 24324 | 5536 | 320 | 8/12/2019 14:25:15 | 8/12/2019 14:25:24 |
| 19951 | sudo | 43712 | 3880 | 75 | 8/12/2019 14:25:15 | 8/12/2019 14:25:15 |

Рисунок 5. Данные, выводимые в консоль при чтении из загружаемого модуля.

Колонки FN(first_noticed) и LU(last_updated) показывают дату и время первого и последнего фиксирования данных о процессе, соответственно. Так, например, информация о процессе 19399 bash была впервые добавлена в список в 14:25:15 восьмого декабря 2019, а последнее обновление данных о нем произошло в 14:25:24 того же дня. Колонка COUNT демонстрирует количество страничных прерываний, подсчитанных модулем для этого процесса. Таким образом, за 9 секунд процесс 19399 вызвал 320 страничных прерываний.

Разумеется, это число не показывает реального положения вещей, так как многие процессы были запущены гораздо раньше модуля, и некоторые страничные прерывания, вызванные такими процессами, не были зафиксированы. Тем не менее число, демонстрируемое в колонке COUNT, в купе с выведеными временными рамками, дает представление о том, насколько часто тот или иной процесс вызывает страничные прерывания.

# ЗАКЛЮЧЕНИЕ

Реализованный загружаемый модуль отвечает техническому заданию. В ходе реализации данного проекта были выполнены поставленные задачи:

1) проанализированы структуры ядра, отвечающие за хранение информации о выделенной процессу памяти;

2) реализован загружаемый модуль ядра, позволяющий собирать информацию о выделении памяти процессам;

3) разработана структура данных для хранения информации о процессах, а также реализован механизм получения этой информации из ядра.

Были выполнены требования, предъявленные к загружаемому модулю:

1) модуль фиксирует вызовы $handle\_mm\_fault()$;

2) при возникновении страничного прерывания, модуль определяет, какой процесс его вызвал и сохраняет информацию о нем в специально разработанную структуру Process;

3) модуль фиксирует время первого и последнего обновления информации о процессе;

4) модуль осуществляет передачу данных из пространства ядра в пространство пользователя с помощью специальной функции ядра, предварительно преобразуя данные списка процессов в понятный пользователю формат.

Полученный модуль позволяет производить мониторинг выделения памяти процессам, а также оценивать число страничных прерываний в единицу времени для каждого конкретного процесса.

# СПИСОК ЛИТЕРАТУРЫ


1. C. Shulyupin «Make Linux Embedded Linux Software, Device Drivers» - [Электронный ресурс] http://www.makelinux.net/ (дата обращения: 18.10.2019)
2. Raghu Bharadwaj «Mastering Linux Kernel Development: A kernel developer's reference manual» - Birmingham: Packt Publishing Ltd - 2017. - 330 стр.
3. Стивенс У.Р. «UNIX. Профессиональное программирование». - Питер, 2018. - 944 с.
4. Bootlin. Linux source code - [Электронный ресурс] https://elixir.bootlin.com/linux/latest/source (дата обращения: 19.10.2019)
5. LWN.net «An introduction to KProbes» - [Электронный ресурс] https://lwn.net/Articles/132196/ (дата обращения: 20.11.2019)
6. О.Цилюрик «Нестандартные сценарии использования модулей ядра» - [Электронный ресурс] https://www.ibm.com/developerworks/ru/library/l-linux_kernel_42/index.html?ca=drs- (дата обращения: 20.11.2019)