



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 ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ**

НА ТЕМУ:

***«Разработка загружаемого модуля ядра Linux для
отслеживания страничных прерываний»***

Студент ИУ7-73Б

_____ Светличная А. А.

Руководитель курсовой работы

_____ Рязанова Н. Ю.

2023 г.

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)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ-7

И. В. Рудаков

«16» сентября 2023 г.

З А Д А Н И Е
на выполнение курсовой работы

по теме

**«Разработка загружаемого модуля ядра Linux
для отслеживания страничных прерываний»**

Студент группы **ИУ7-73Б**

Светличная Алина Алексеевна

Направленность КР

учебная

Источник тематики

НИР кафедры

График выполнения КР: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание

*Разработать загружаемый модуль ядра Linux для отслеживания страничных прерываний с
возможностью просмотра собранной информации в режиме пользователя.*

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на **12-20** листах формата А4.

Дата выдачи задания «16» сентября 2023 г.

Руководитель курсовой работы

(Подпись, дата)

Рязанова Н. Ю.

(Фамилия И. О.)

Студент

(Подпись, дата)

Светличная А. А.

(Фамилия И. О.)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 Аналитическая часть	5
1.1 Постановка задачи	5
1.2 Структуры ядра связанные с управлением памятью, выделенной процессу	5
1.3 Страничные прерывания	7
1.4 Перехват страничного прерывания	11
1.5 Передача информации из режима ядра в режим пользователя	12
2 Конструкторская часть	14
2.1 IDEF0-диаграмма нулевого уровня	14
2.2 Структура, хранящая информацию об обработанных процессах	14
2.3 Алгоритма обработчика	15
2.4 Алгоритм передачи информации	17
2.5 Структура программного обеспечения	17
3 Технологическая часть	19
3.1 Выбор языка и среды программирования	19
3.2 Реализация загружаемого модуля	19
4 Исследовательская часть	24
4.1 Демонстрация работы модуля	24
ЗАКЛЮЧЕНИЕ	25
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	26
ПРИЛОЖЕНИЕ А	27

ВВЕДЕНИЕ

Страничные прерывания (page faults) в операционных системах, включая Linux, возникают при попытке обращения к виртуальной памяти, которая в данный момент не находится в физической памяти. Это может произойти, например, когда процесс обращается к странице памяти, которая была выгружена на диск в результате оптимизации использования физической памяти или при первом обращении к области памяти, которая еще не была загружена в оперативную память.

Когда возникает страничное прерывание, операционная система должна обработать его, чтобы загрузить соответствующую страницу из дискового хранилища в оперативную память или выполнить другие действия в зависимости от конкретной ситуации. Обработка страничных прерываний является критическим аспектом работы виртуальной памяти в операционных системах, поскольку эффективное управление страничными прерываниями напрямую влияет на производительность системы.

Разработка инструментов для отслеживания страничных прерываний позволяет анализировать их, что полезно для оптимизации работы операционной системы.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра Linux для отслеживания страничных прерываний и определения объемов физической и виртуальной памяти, выделенной процессу.

Для решения поставленной задачи необходимо задач:

- проанализировать структуры, связанные с функциями управлением физической и виртуальной памятью;
- проанализировать функции ядра, связанные со страничными прерываниями;
- разработать алгоритм отслеживания страничных прерываний и сохранения соответствующих данных;
- разработать алгоритм передачи полученного результата из пространства ядра в пространство пользователя;
- разработать общую структуру программного обеспечения;
- реализовать загружаемый модуль в соответствии с разработанными алгоритмами;
- провести исследование работы разработанного программного обеспечения.

1.2 Структуры ядра связанные с управлением памятью, выделенной процессу

Физическая память в операционной системе Linux разделена на отдельные модули, называемые страницами. Значительная часть внутренней системной обработки памяти производится на постраничной основе. Размер страницы варьируется от одной архитектуры к другой, хотя большинство систем в настоящее время использует страницы по 4096 байт (в данной работе

будут рассмотрен именно такой размер страниц). Постоянная `PAGE_SIZE` задаёт размер страницы для любой архитектуры [1].

Виртуальная память, выделяемая процессу, разделена на области VMA - virtual memory area, описывающие участки адресного пространства процесса

Процессы в Linux представлены структурой `task_struct`. Одно из полей данной структуры `struct mm_struct *mm` является указателем на структуру `mm_struct`, которая ранее содержала указатель на голову списка областей VMA `vm_area_struct *mmap` [2]. Однако в современной версии (6.7.4) это поле отсутствует. Структура `vm_area_struct` представлена в листинге 1.1.

Листинг 1.1 – Структура `vm_area_struct`

```
1 struct vm_area_struct {
2     union {
3         struct {
4             unsigned long vm_start;
5             unsigned long vm_end;
6         };
7     };
8     ...
9 }
```

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, ранее передвижение между ними было возможно за счет указателей `vm_next`, `vm_prev`, однако сейчас данные поля отсутствуют, данная возможность поддерживается макросом `VMA_ITERATOR`, синопсис которого показан в листинге 1.2.

Листинг 1.2 – Синопсис макроса `VMA_ITERATOR`

```
1 #define VMA_ITERATOR(name, __mm, __addr)
2     struct vma_iterator name = {
3         .mas = {
4             .tree = &(__mm)->mm_mt,
5             .index = __addr,
```

```

6         .node = MAS_START,
7     },
8 }

```

Физическая память, описывается структурой `struct mm_rss_stat rss_stat`, представлена в листинге 1.3, поле `count` хранит счетчик выделенных процессу физических страниц.

Листинг 1.3 – Структура `mm_rss_stat`

```

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

```

Получить информацию, хранящуюся в данной структуре можно с помощью функции `get_mm_rss(struct mm_struct *mm)`, представленной в листинге 1.4.

Листинг 1.4 – Функция `get_mm_rss`

```

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 }

```

Данная функция возвращает суммарное число страниц, выделенных процессу.

- `MM_FILEPAGES` — страницы, выделенные для файлов процесса;
- `MM_ANONPAGES` — анонимные страницы;
- `MM_SHMEMPAGES` — разделяемые страницы.

1.3 Страничные прерывания

Страничные прерывания в Linux возникают при обращении к странице, которая не находится в физической оперативной памяти (RAM). Этот процесс инициируется когда процессор обнаруживает отсутствие запрашиваемой

страницы в текущем наборе страниц, находящихся в оперативной памяти. Однако это не означает, что данное явление возникает только в результате ошибки, это может означать еще и запрос на выделение страницы [3]. Ниже представлена схема определения причины возникновения страничного прерывания обработчиком.

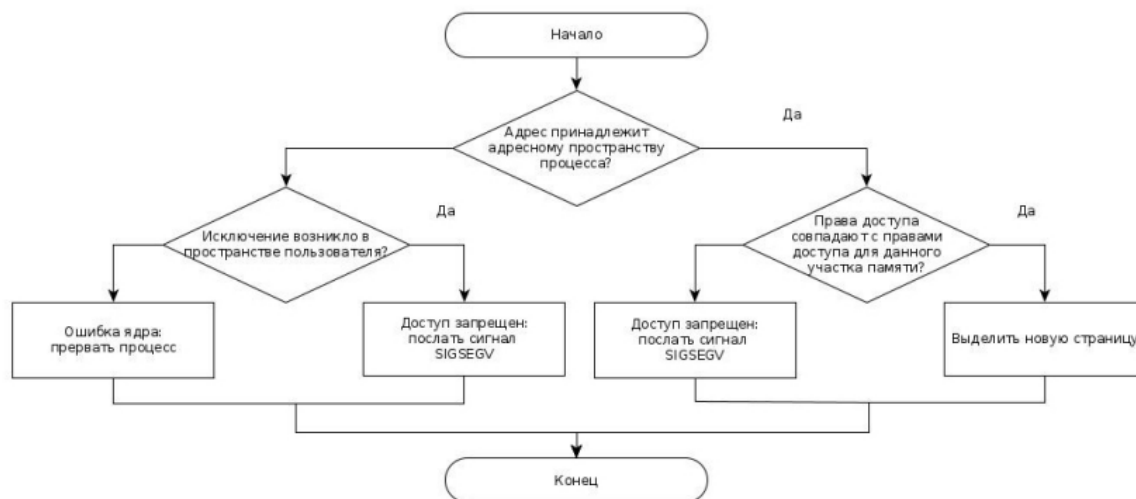


Рисунок 1.1 – Частичная схема определения причины страничного прерывания

Однако данная схема является упрощенной, полная схема на английском языке изображена на рисунке 1.2.

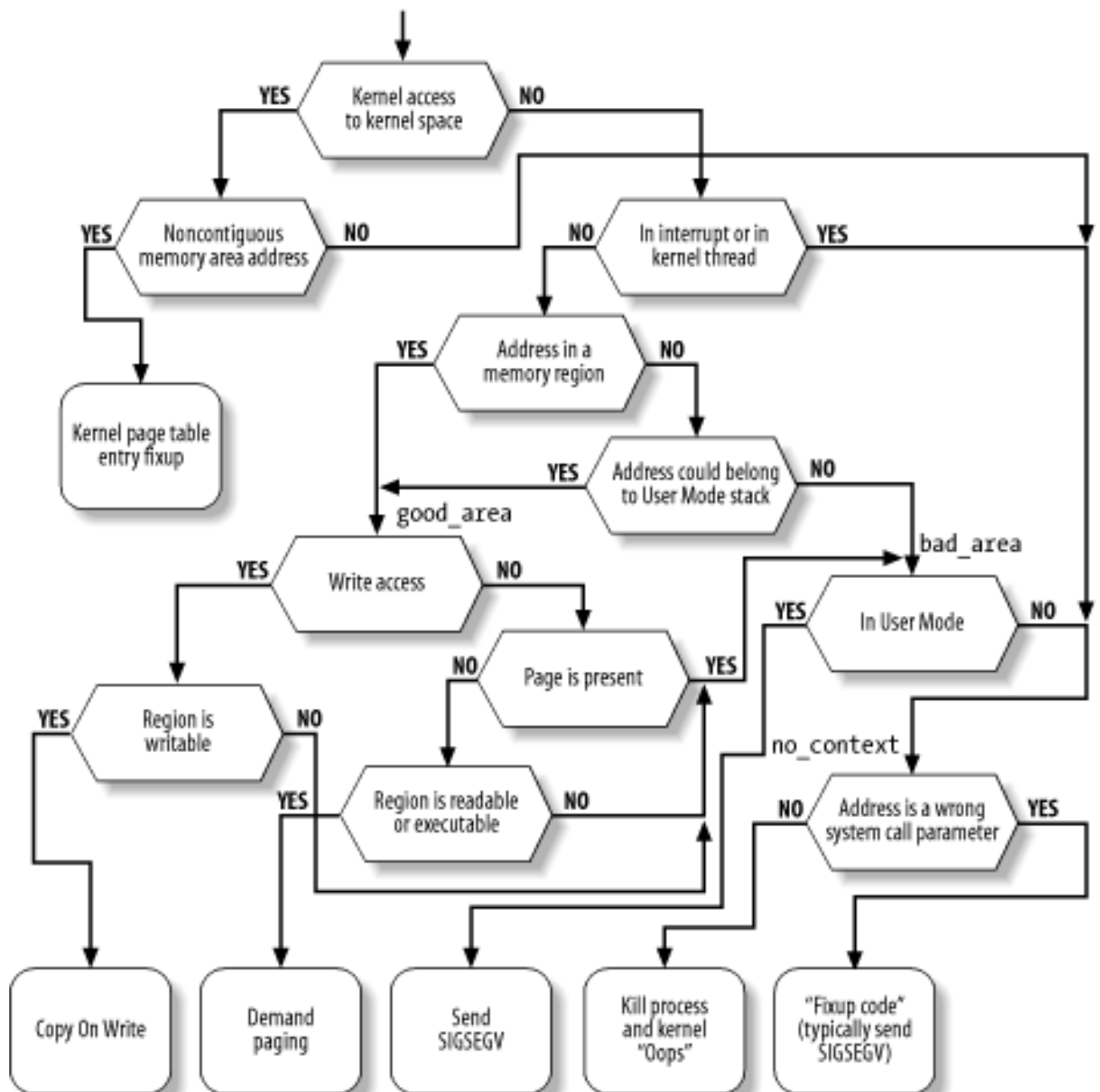


Рисунок 1.2 – Полная схема определения причины страничного прерывания

Обработка страничного прерывания начинается с функции `do_page_fault`, листинг которой показан ниже.

Листинг 1.5 – Функция `handle_mm_fault`

```

1 asmlinkage void
2 do_page_fault(unsigned long address, unsigned long mmcsr,
3               long cause, struct pt_regs *regs)
4 {
5     struct vm_area_struct * vma;
6     struct mm_struct *mm = current->mm;

```

```

7         vm_fault_t fault;
8     ...
9
10        fault = handle_mm_fault(vma, address, flags, regs);
11    ...
12
13        if (unlikely(fault & VM_FAULT_ERROR)) {
14            if (fault & VM_FAULT_OOM)
15                goto out_of_memory;
16            else if (fault & VM_FAULT_SIGSEGV)
17                goto bad_area;
18            else if (fault & VM_FAULT_SIGBUS)
19                goto do_sigbus;
20            BUG();
21        }
22    ...
23 }

```

Как видно из исходного кода функции `do_page_fault`, основным обработчиком страничного прерывания является `handle_mm_fault`, внутри которой происходит выделения памяти и проверка отсутствия ошибок выделения.

Листинг 1.6 – Функция `handle_mm_fault`

```

1 int handle_mm_fault(struct task_struct *tsk,
2                     struct vm_area_struct * vma,
3                     unsigned long address,
4                     int write_access)
5 {
6     pgd_t *pgd;
7     pmd_t *pmd;
8
9     pgd = pgd_offset(vma->vm_mm, address);
10    pmd = pmd_alloc(pgd, address);
11    if (pmd) {
12        pte_t * pte = pte_alloc(pmd, address);

```

```

13         if (pte) {
14             if (handle_pte_fault(tsk, vma, address, write_access
15                 ↪ , pte)) {
16                 update_mmu_cache(vma, address, *pte);
17                 return 1;
18             }
19         }
20
21     return 0;
22 }

```

Таким образом, чтобы отследить возникновение страничного прерывания, необходимо обнаружить вызов `handle_mm_fault`.

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` на текущий процесс [4]. Таким образом, при возникновении страничного прерывания, можно получить информацию о процессе, вызвавшем его (`pid`, `comm` и т.д.).

1.4 Перехват страничного прерывания

Все системные вызовы проходят через таблицу `sys_call_table`. Индекс обработчика соответствует номеру системного вызова. Для отслеживания страничного прерывания возможно заменить существующий системный вызов в данной таблице своим. Однако такой подход сложен и небезопасен, поскольку требуется полностью переписать обработчик, что при неправильном использовании может привести к нарушению работы системы. Кроме того, необходимо хранить изначальный обработчик, чтобы иметь возможность его восстановить при выгрузке модуля [5].

Kprobe представляет собой механизм в ядре операционной системы Linux, позволяющий динамически встраивать точки прерывания в код ядра или модулей ядра для отладки и сбора данных о выполнении программы. Пользовательское приложение или модуль ядра может зарегистрировать «пробу», указывая адрес функции или инструкции, в которой необходимо установить точку прерывания. Когда установленная точка прерывания достигается во

время выполнения программы, ядро встраивает код обработчика прерывания в соответствующее место. Можно установить предобработчик, выполняющийся до инструкции, или постобработчик, выполняющийся после [6].

Kprobe описывается структурой, представленной в листинге 1.7:

Листинг 1.7 – Структура kprobe

```
1 struct kprobe {
2     /* Allow user to indicate symbol name of the probe point */
3     const char *symbol_name;
4
5     kprobe_pre_handler_t pre_handler;
6     kprobe_post_handler_t post_handler;
7     ...
8 };
```

Данный метод не требует внесения изменений в ядро, что делает его более предпочтительным.

1.5 Передача информации из режима ядра в режим пользователя

В Linux передача информации из режима ядра в режим пользователя часто осуществляется через виртуальную файловую систему /proc. /proc предоставляет интерфейс для доступа к информации о текущем состоянии ядра, процессов и других системных ресурсов в виде файлов и каталогов. Этот механизм обеспечивает простой и удобный способ для программ в пользовательском пространстве получать доступ к различным данным, которые управляются ядром.

Для созданного файла с помощью структуры `proc_ops` можно определить собственную функцию чтения использованием специальной функции `copy_to_user`.

Выводы

В результате проведенного анализа определены структуры ядра, содержащие информацию о процессе (`struct task_struct`), его виртуальном адресном пространстве (`struct vm_area_struct`) и физической памяти, выде-

ленной процессу (`struct mm_rss_stat`). Определены этапы обработки страничного прерывания на базе страничного исключения. Страничное исключение обрабатывается функцией `handle_mm_fault`. Выбран механизм регистрации собственного обработчика — `kprobe`. Определен механизм передачи информации из режима ядра в режим пользователя — виртуальная файловая система `/proc`.

2 Конструкторская часть

2.1 IDEF0-диаграмма нулевого уровня

На рисунке 2.1 представлена IDEF0-диаграмма отслеживания страничных прерываний.

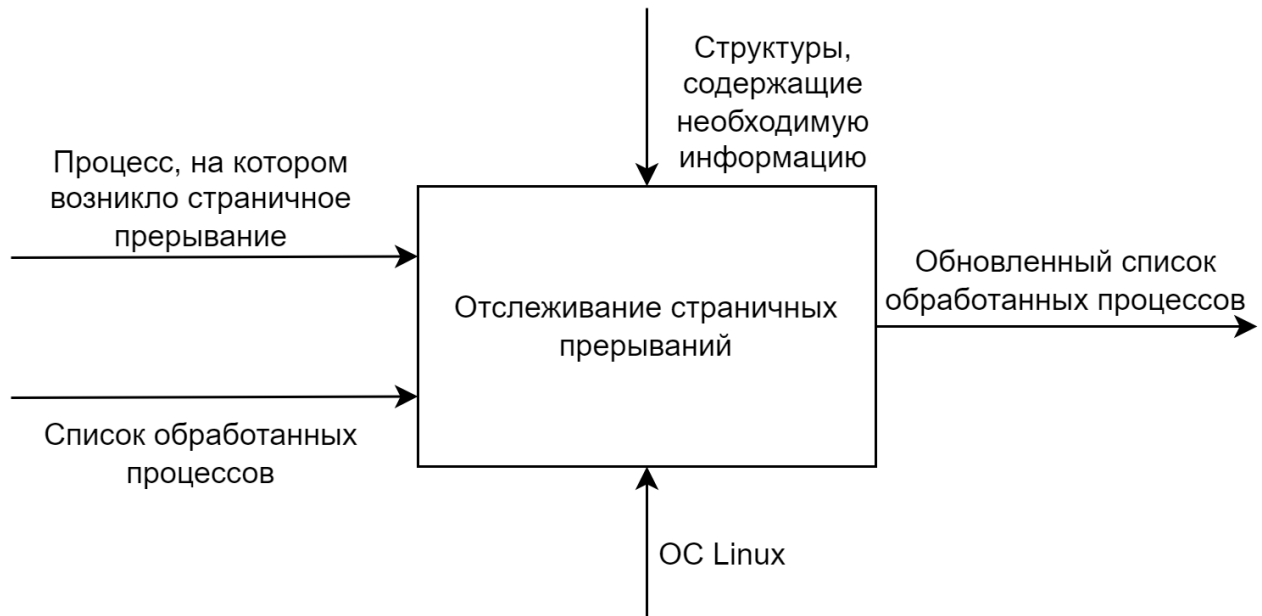


Рисунок 2.1 – IDEF0-диаграмма отслеживания страничных прерываний

2.2 Структура, хранящая информацию об обработанных процессах

Для отслеживания страничных прерывания и выделенной процессу памяти необходима структура, имеющая следующие поля:

- идентификатор процесса;
- имя процесса;
- объем выделенной физической памяти;
- объем выделенной виртуальной памяти;
- количество страничных прерываний, возникших с момента загрузки модуля.

Если на процессе ранее возникало страничное прерывание, то необходимо обновлять имеющуюся информацию, то есть нужно запоминать все процессы, в которых возникало страничное прерывание. Для этого удобно использовать односвязный список, неограничивающий количество возможно обработанных процессов. То есть при очередном вызове обработчика в первую очередь необходимо проверить наличие данного процесса в списке, и если процесс не найден, то создать новый элемент.

2.3 Алгоритма обработчика

На рисунке 2.2 представлена схема собственного обработчика страничного прерывания.

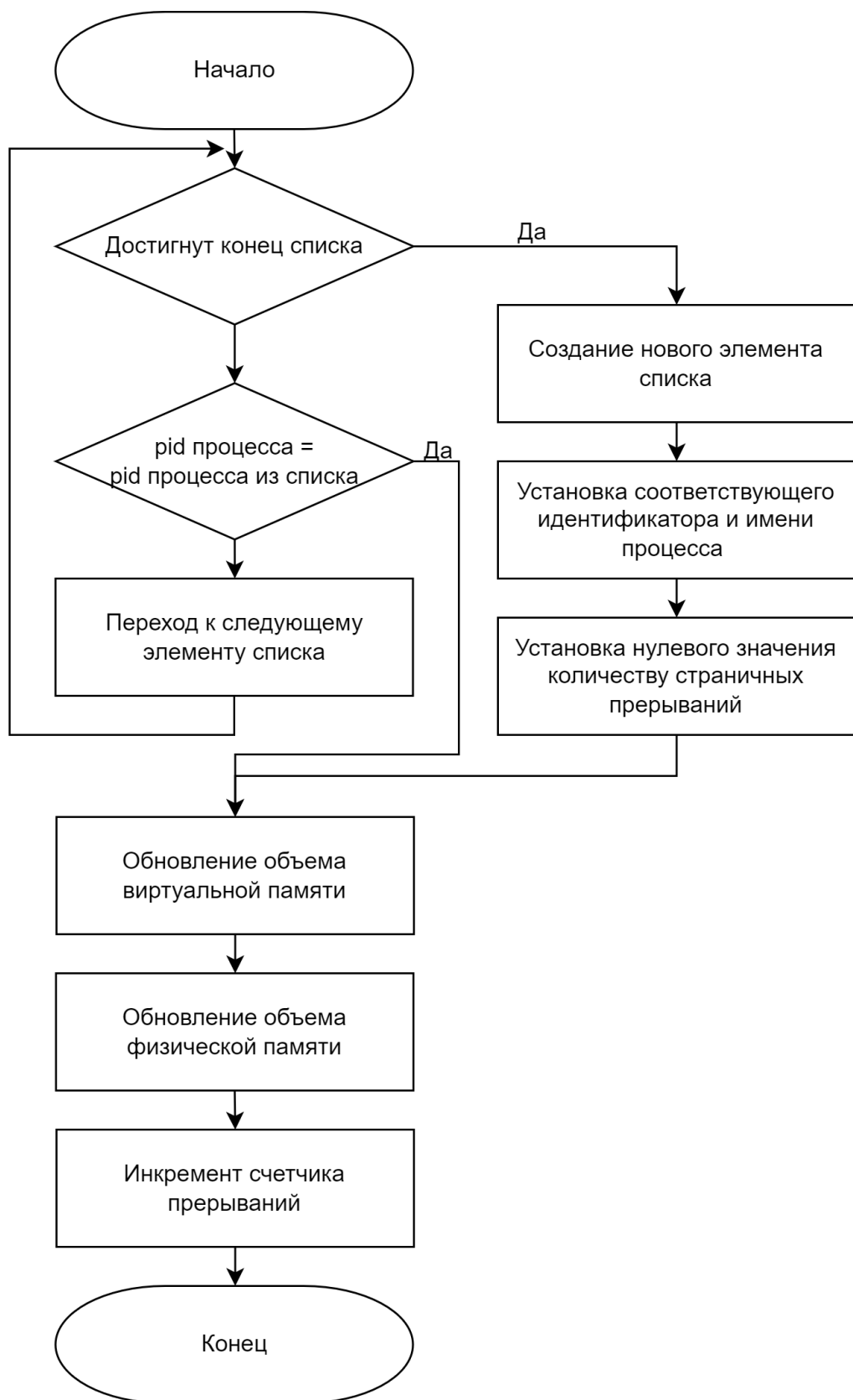


Рисунок 2.2 – Схема алгоритма обработки прерывания

2.4 Алгоритм передачи информации

Данные, полученные во время мониторинга записываются в связный список. Для вывода информации в режиме пользователя, полученной результат необходимо перевести в текстовое представление, для чего следует реализовать функцию `create_buffer`. Данная функция должна принимать список и составлять из него сообщение, которое затем необходимо передать в пространство пользователя.

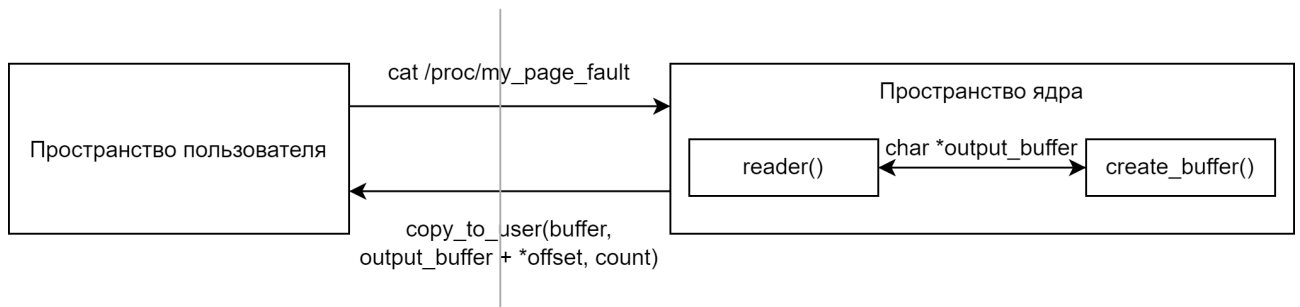


Рисунок 2.3 – Схема передачи результата работы из пространство ядра в пространство пользователя

2.5 Структура программного обеспечения

На рисунке 2.4 изображена структура ПО.

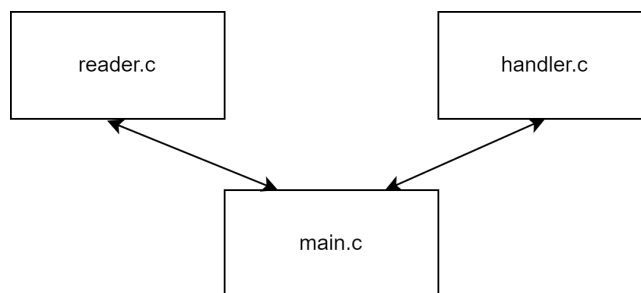


Рисунок 2.4 – Структура программного обеспечения

В данной структуре есть три модуля, отвечающие за:

- `main.c` — загрузка и выгрузка ядра, инициализация и регистрация основных структур и механизмов;
- `handler.c` — перехват страничного прерывания и запись необходимой информации;

- `reader.c` — конвертация результата в удобочитаемый вид и передача его из режима ядра в режим пользователя.

Однако данные модули являются достаточно маленькими, поэтому можно их объединить в один.

3 Технологическая часть

3.1 Выбор языка и среды программирования

В качестве языка программирования был выбран C. По той причине, что именно этот язык включает в себя большое количество библиотек, без которых невозможно реализовать поставленную задачу.

В качестве средства сборки была выбрана утилита `make`, так как она облегчает компиляцию и цнифицирует ее.

3.2 Реализация загружаемого модуля

Реализация загружаемого модуля содержит следующие функции:

- `init_page_fault` — инициализация модуля, содержащая создания файла `/proc` и регистрацию `kprobe`;
- `handler` — обработчик страничного прерывания;
- `find_process_by_id` — определение элемента списка процессов с таким же `pid`, как процесс, вызвавший прерывание;
- `create_process` — если данный процесс не находится в списке, то функция вызывается для выделения памяти под структуру и ее поля, а также постановки нового элемента в список;
- `count_vma` — вычисляет объем виртуальной памяти процесса;
- `count_rss` — вычисляет объем физической памяти процесса;
- `reader` — функция вызывается при чтении файла `/proc/my_page_fault`;
- `create_buffer` — переводит результат из односвязного списка в текстовое представление;
- `exit_page_fault` — вызывается при выгрузке модуля;
- `free_processes` — освобождает память из-под структур списка и их полей.

Основными функциями можно назвать `handler` и `find_process_by_id` как функции обработки страничных прерываний; `reader` и `create_buffer` как функции передачи результата из пространства ядра в пространство пользователя. Листинги данных частей кода показаны ниже.

Листинг 3.1 – Листинг основных функций обработки страничных прерываний

```
1 struct Process *find_process_by_id(struct Process *head, long
   ↪ pid){
2     struct Process *cur = head;
3
4     while (cur != NULL)
5     {
6         if (cur->pid == pid)
7             break;
8
9         cur = cur->next;
10    }
11
12    return cur;
13 }
14
15 void handler(struct kprobe *p, struct pt_regs *regs, unsigned
   ↪ long flags) {
16     struct Process *process = find_process_by_id(head, current
   ↪ ->pid);
17
18     if (process == NULL) {
19         process = create_process(&head, process);
20         process->pid = current->pid;
21         strncpy(process->comm, current->comm, COMM_LEN);
22         process->page_fault_counter = 0;
23     }
24
25     process->vma = count_vma(current);
```

```

26     process->rss = count_rss(current);
27     process->page_fault_counter++;
28 }

```

Листинг 3.2 – Листинг передачи результата из пространство ядра в пространство пользователя

```

1 char *create_buffer(char *output_buffer)
2 {
3     char *tmp = vmalloc(KB), *separator = vmalloc(COMM_LEN);
4
5     memset(tmp, 0, KB);
6     memset(output_buffer, 0, KB);
7
8     sprintf(output_buffer, "PID\tCOMM\t\t VMA/KB\tRSS/KB\tPF\n"
9         ↪ );
10
11     struct Process *cur = head;
12
13     while (cur != NULL)
14     {
15         memset(separator, 0, COMM_LEN);
16         memset(separator, ' ', COMM_LEN - strlen(cur->comm));
17
18         sprintf(tmp, "%ld\t%s%s%ld\t%ld\t%ld\t\n",
19             cur->pid,
20             cur->comm,
21             separator,
22             cur->vma,
23             cur->rss,
24             cur->page_fault_counter);
25
26         strcat(output_buffer, tmp);
27
28         cur = cur->next;

```

```

28     }
29
30     vfree(tmp);
31     vfree(separator);
32
33     return output_buffer;
34 }
35
36 static ssize_t reader(struct file *file, char __user *buffer,
    ↪ size_t count, loff_t *offset)
37 {
38     char *output_buffer = vmalloc(KB);
39     output_buffer = create_buffer(output_buffer);
40
41     ssize_t len = strlen(output_buffer);
42     if (*offset >= len)
43         return 0;
44
45     if (*offset + count > len)
46         count = len - *offset;
47
48     if (copy_to_user(buffer, output_buffer + *offset, count) !=
    ↪ 0)
49         return -EFAULT;
50
51     *offset += count;
52
53     vfree(output_buffer);
54
55     return count;
56 }
57
58 static const struct proc_ops proc_fops = {
59     .proc_read = reader,

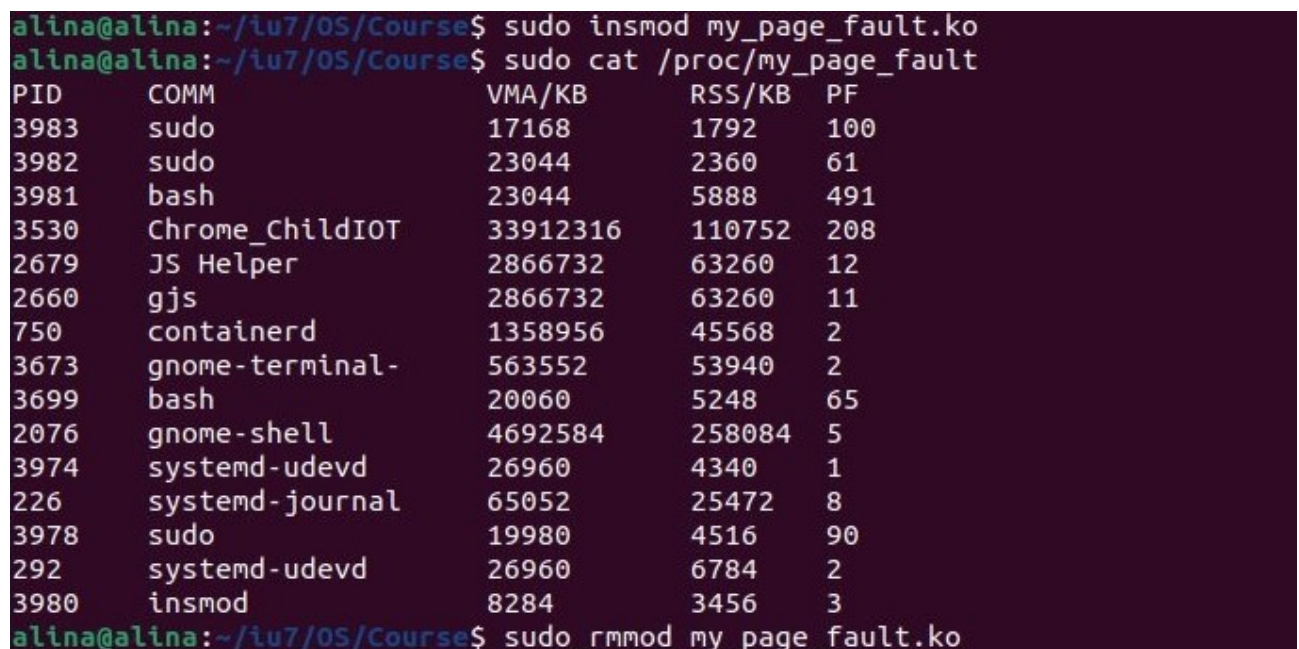
```

60 };

4 Исследовательская часть

4.1 Демонстрация работы модуля

На рисунке 4.1 представлена демонстрация работоспособности приложения.



```
alina@alina:~/iu7/OS/Course$ sudo insmod my_page_fault.ko
alina@alina:~/iu7/OS/Course$ sudo cat /proc/my_page_fault
```

PID	COMM	VMA/KB	RSS/KB	PF
3983	sudo	17168	1792	100
3982	sudo	23044	2360	61
3981	bash	23044	5888	491
3530	Chrome_ChildIOT	33912316	110752	208
2679	JS Helper	2866732	63260	12
2660	gjs	2866732	63260	11
750	containerd	1358956	45568	2
3673	gnome-terminal-	563552	53940	2
3699	bash	20060	5248	65
2076	gnome-shell	4692584	258084	5
3974	systemd-udevd	26960	4340	1
226	systemd-journal	65052	25472	8
3978	sudo	19980	4516	90
292	systemd-udevd	26960	6784	2
3980	insmod	8284	3456	3

```
alina@alina:~/iu7/OS/Course$ sudo rmmod my_page_fault.ko
```

Рисунок 4.1 – Результат работы разработанного загружаемого модуля

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы был реализован загружаемый модуль ядра Linux для отслеживания страничных прерываний и объемов физической и виртуальной памяти выделенной процессу.

Проанализированы структуры, содержащие информацию о процессе (`struct task_struct`), его виртуальном адресном пространстве (`struct vm_area_struct`) и физической памяти, выделенной процессу (`struct mm_rss_stat`).

Проанализированы функции ядра, связанные со страничными прерываниями. Выделено, что страничное исключение обрабатывается функцией `handle_mm_fault`.

Проанализированы механизмы регистрации собственного обработчика прерываний. Выбран `kprobe`, так как для него не требуется вносить изменения в ядро.

Проанализированы механизмы передачи информации из пространство ядра в пространство пользователя. Выбрана виртуальная файловая система `/proc`.

Разработаны алгоритм отслеживания страничных прерываний и сохранения соответствующих данных и алгоритм передачи полученного результата из пространства ядра в пространство пользователя.

Разработана структура программного обеспечения, согласно которой реализован загружаемый модуль в соответствии с разработанными ранее алгоритмами.

Проведено исследование разработанного программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Управление памятью в Linux [Электронный ресурс]. URL: https://dmilvdv.narod.ru/Translate/LDD3/ldd_memory_management_linux.html (дата обращения: 01.02.2024).
2. ПРОЦЕССЫ В LINUX [Электронный ресурс]. URL: https://www.opennet.ru/base/dev/proccess_in_linux.txt.html (дата обращения: 01.02.2024).
3. Understanding and troubleshooting page faults and memory swapping [Электронный ресурс]. URL: <https://www.site24x7.com/learn/linux/page-faults-memory-swapping.html> (дата обращения: 01.02.2024).
4. Current Process [Электронный ресурс]. URL: <https://tuxthink.blogspot.com/2011/04/current-process.html> (дата обращения: 10.02.2024).
5. System Calls [Электронный ресурс]. URL: <https://tldp.org/LDP/lkmpg/2.4/html/x939.html> (дата обращения: 09.02.2024).
6. Concepts: Kprobes and Return Probe [Электронный ресурс]. URL: <https://docs.kernel.org/trace/kprobes.html> (дата обращения: 09.02.2024).

ПРИЛОЖЕНИЕ А

Листинг 4.1 – Листинг реализованного загружаемого модуля для
отслеживания страничных прерываний

```
1 #include <linux/proc_fs.h>
2 #include <linux/kprobes.h>
3 #include <linux/vmalloc.h>
4
5 MODULE_LICENSE("GPL");
6
7 #define PROC_ENTRY_NAME "my_page_fault"
8
9 #define KB 1024
10
11 #define COMM_LEN 20
12
13 struct Process
14 {
15     long pid;
16     char *comm;
17     long vma;
18     long rss;
19     long page_fault_counter;
20     struct Process *next;
21 };
22
23 struct Process *head = NULL;
24
25 struct Process *find_process_by_id(struct Process *head, long
    ↪ pid){
26     struct Process *cur = head;
27
28     while (cur != NULL)
```

```

29     {
30         if (cur->pid == pid)
31             break;
32
33         cur = cur->next;
34     }
35
36     return cur;
37 }
38
39 struct Process *create_process(struct Process **head, struct
    ↪ Process *new_process) {
40     new_process = vmalloc(sizeof(struct Process));
41     new_process->comm = vmalloc(COMM_LEN);
42     new_process->next = NULL;
43
44     if (*head == NULL) {
45         *head = new_process;
46         (*head)->next = NULL;
47     }
48     else {
49         new_process->next = *head;
50         *head = new_process;
51     }
52
53     return new_process;
54 }
55
56 long count_vma(struct task_struct *task) {
57     long virt_mem = 0;
58     struct vm_area_struct *vma;
59
60     VMA_ITERATOR(iter, task->mm, 0);
61

```

```

62     for_each_vma(iter, vma)
63         virt_mem += vma->vm_end - vma->vm_start;
64
65     return virt_mem / KB;
66 }
67
68 long count_rss(struct task_struct *task){
69     return get_mm_rss(task->mm) * PAGE_SIZE / KB;
70 }
71
72 void handler(struct kprobe *p, struct pt_regs *regs, unsigned
    ↪ long flags) {
73     struct Process *process = find_process_by_id(head, current
    ↪ ->pid);
74
75     if (process == NULL) {
76         process = create_process(&head, process);
77         process->pid = current->pid;
78         strncpy(process->comm, current->comm, COMM_LEN);
79         process->page_fault_counter = 0;
80     }
81
82     process->vma = count_vma(current);
83     process->rss = count_rss(current);
84     process->page_fault_counter++;
85
86     //printk(KERN_INFO "PID: %ld, Comm: %s, VMA/KB: %ld, RSS/KB
    ↪ : %ld, Page Faults: %ld\n",
87         //process->pid, process->comm,
88         //process->vma, process->rss,
89         //process->page_fault_counter);
90 }
91
92 char *create_buffer(char *output_buffer)

```

```

93 {
94     char *tmp = vmalloc(KB), *separator = vmalloc(COMM_LEN);
95
96     memset(tmp, 0, KB);
97     memset(output_buffer, 0, KB);
98
99     sprintf(output_buffer, "PID\tCOMM\t\t VMA/KB\tRSS/KB\tPF\n"
    ↪ );
100
101     struct Process *cur = head;
102
103     while (cur != NULL)
104     {
105         memset(separator, 0, COMM_LEN);
106         memset(separator, ' ', COMM_LEN - strlen(cur->comm));
107
108         sprintf(tmp, "%ld\t%s%s%ld\t%ld\t%ld\t\n",
109                 cur->pid,
110                 cur->comm,
111                 separator,
112                 cur->vma,
113                 cur->rss,
114                 cur->page_fault_counter);
115
116         strcat(output_buffer, tmp);
117
118         cur = cur->next;
119     }
120
121     vfree(tmp);
122     vfree(separator);
123
124     return output_buffer;
125 }

```

```

126
127 static ssize_t reader(struct file *file, char __user *buffer,
    ↪ size_t count, loff_t *offset)
128 {
129     char *output_buffer = vmalloc(KB);
130     output_buffer = create_buffer(output_buffer);
131
132     ssize_t len = strlen(output_buffer);
133     if (*offset >= len)
134         return 0;
135
136     if (*offset + count > len)
137         count = len - *offset;
138
139     if (copy_to_user(buffer, output_buffer + *offset, count) !=
    ↪ 0)
140         return -EFAULT;
141
142     *offset += count;
143
144     vfree(output_buffer);
145
146     return count;
147 }
148
149 static const struct proc_ops proc_fops = {
150     .proc_read = reader,
151 };
152
153 static struct kprobe kp = {
154     .symbol_name = "__handle_mm_fault",
155     .post_handler = handler,
156 };
157

```

```

158 static int __init init_page_fault(void) {
159     proc_create(PROC_ENTRY_NAME, 0, NULL, &proc_fops);
160     register_kprobe(&kp);
161
162     printk(KERN_INFO "Module initialized\n");
163     return 0;
164 }
165
166 void free_processes(struct Process *head){
167     struct Process *cur = head;
168
169     while (cur != NULL)
170     {
171         vfree(cur->comm);
172         vfree(cur);
173         cur = cur->next;
174     }
175 }
176
177 static void __exit exit_page_fault(void) {
178     free_processes(head);
179     unregister_kprobe(&kp);
180     remove_proc_entry(PROC_ENTRY_NAME, NULL);
181
182     printk(KERN_INFO "Module exited\n");
183 }
184
185 module_init(init_page_fault);
186 module_exit(exit_page_fault);

```