1.

Содержание

Введение

В настоящее время большую актуальность приобретает исследования выделения памяти многопоточным приложениям. При этом отдельно рассматривают проблемы выделения виртуальной и физической памяти при выполнении процесса.

Целью данной работы является разработка загружаемого модуля ядра, который будет предоставлять возможность пользователю мониторинга виртуальной памяти и анализа количества выделенных страниц.

В данной работе также приведено исследование выделения виртуальной памяти процессам в зависимости от количества запрашиваемой памяти и от количества выполняемых потоков.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу по курсу Операционные системы необходимо разработать загружаемый модуль ядра, предоставляющий пользователю возможность получения информации о виртуальной памяти процесса. Также будет произведено исследование выделения виртуальной памяти многопоточным приложениям на основе анализируемой программы, которая тоже будет реализованна в данном курсовом проекте.

Для решения задачи необходимо.

а) Определить структуры, связанные с поставленной задачей.

б) Проанализировать и выбрать способ получения информации о виртуальной памяти процесса.

в) Разработать алгоритм определения количества выделяемых процессу страниц по запросам на выделение памяти.

г) Разработать структуру ПО.

д) Реализовать ПО.

е) Провести исследования выделения памяти.

1.1.1 Анализ выделения памяти процессу

Стали делить память на страницы. Можно выполнить программу, которая находится не целиком в памяти. Для этого нужно содержать части кода с которыми в текущий момент работает процессор. Это воплотилось в понятие виртуальная память.

Виртуальная память – память, размер которой превышает размер реального физического пространства. Виртуальная память сама по себе ничего не хранит. Виртуальное адресное пространство — это абстракция, но оно определенным образом поставлено в соответствие физической памяти.

Загрузка частей программы в память выполняется по запросу. Т.е. соответствие части кода загружаемого по запросу, когда процессор обращается к этим частям кода.

1.1.2 Таблица страниц

Адресное пространство процесса и адресное пространство физической памяти делится на блоки равного размера. Блоки, на которые делится адресное пространство процесса называют страницами, а блоки на которые делится физическая память – кадрами, фреймами или блоками.

Виртуальный адрес состоит из двух частей:

— p - номер страницы,

— d - смещение страницы.

На рис. 1.1 продемонстрировано отображение виртуальной памяти на физическую с помощью таблицы страниц.

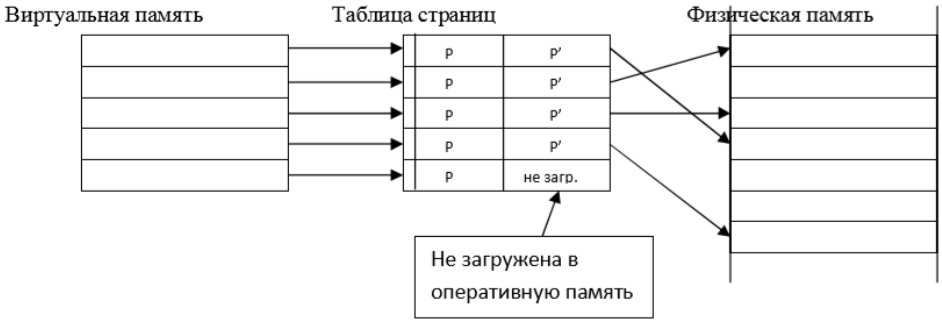

Рисунок 1.1 — Отображение виртуальной памяти на физическую с помощью таблицы страниц

С помощью таблиц страниц процессор осуществляет преобразование виртуального адреса в физический. У каждого процесса есть свой набор таблиц страниц. Как только происходит переключение процесса (context switch), меняются и таблицы страниц. В Linux, указатель на таблицы страниц процесса хранится в поле pgd дескриптора памяти процесса. Каждой виртуальной странице соответствует одна запись в таблице страниц.

На рис. 1.2 показана 4-байтовая запись pgd.

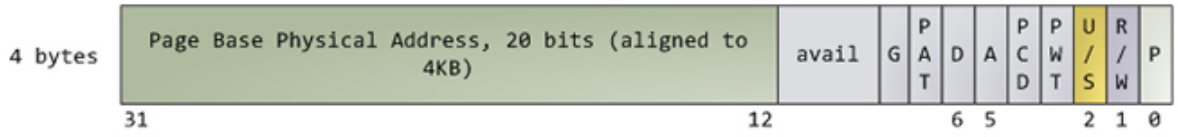

Рисунок 1.2 — 4-байтовая запись pgd

Флаг «P» говорит о том, находится ли страница в оперативной памяти или нет. Когда данный флаг установлен в 0, доступ к соответствующей странице вызовет page fault. Флаг «R/W» означает «запись/чтение»; если флаг не установлен, то к странице возможен доступ только на чтение. Флаг «U/S» означает «пользователь/супервайзер»; если флаг не установлен, только код выполняющийся с уровнем привилегий 0 (т.е. ядро) может обратиться к данной странице. Таким образом, данные флаги используются для того, чтобы реализовать концепцию адресного пространства доступного только на запись и пространства, которое доступно только для ядра. Флаги «D» и «A» означают «dirty» и «accessed». «Dirty-страница» – эта та, в которую была недавно проведена запись, а «accessed»-страница – это страница, к которой было осуществлено обращение (чтение или запись). pgd хранит начальный физический адрес страницы в памяти.

При выполнении программы, которая находится не целиком в памяти, процесс потребует страницу, которой нет в оперативной памяти - возникнет исключение (страничная неудача - исправимое исключение), которое будет обработано в режиме ядра. В результате менеджер памяти попытается загрузить страницу в свободную память, а процесс на это время будет заблокирован. По завершении работы менеджера памяти страница будет загружена и процесс будет продолжать выполнятся с той команды, на которой возникло исключение. Если свободная страница в физической памяти отсутствует, то менеджер памяти должен выбрать страницу для замещения.

Процесс может обращаться только к разрешенным областям памяти. Каждой области памяти назначаются определенные права доступа, такие как чтение, запись или выполнение, которые процесс должен неукоснительно соблюдать. Если процесс обращается по адресу, который не относится к разрешенной области памяти, или если доступ к разрешенной области памяти выполняется некорректным образом,

ядро уничтожает такой процесс с сообщением «Segmentation Fault» (Ошибка сегментации).

В областях памяти может содержаться вся нужная процессу информация, такая как:

— машинный код, загруженный из исполняемого файла в область памяти процесса, которая называется сегментом кода (text section);

— инициализированные переменные, загруженные из исполняемого файла в область памяти процесса, которая называется сегментом данных (data section);

— страницы памяти, заполненные нулями, в которых содержатся неинициализированные глобальные переменные программы. Эта область памяти называется сегментом bss 1 (bss section);

— страницы памяти, заполненные нулями, в которых находится пользовательский стек процесса;

— дополнительные сегменты кода, данных и BSS для каждой совместно используемой библиотеки, такой как библиотека libc и динамический компоновщик, которые загружаются в адресное пространство процесса.

1.2 Структуры ядра

1.2.1 Структура task_struct

Список процессов хранится в ядре в виде циклического двухсвязного списка, который называется списком задач (task list). Каждый элемент этого списка описывает один запущенный процесс и называется дескриптором процесса. Дескриптор процесса имеет тип task_struct, структура которого описана в файле <linux/sched.h>. Дескриптор процесса содержит всю информацию об определенном процессе. В дескрипторе процесса содержатся данные, которые описывают выполняющуюся программу, — открытые файлы, адресное пространство процесса, сигналы, ожидающие обработки, состояние процесса и многое другое (рис. 1.3). На листинге 1.1 представлена часть структуры task_struct.

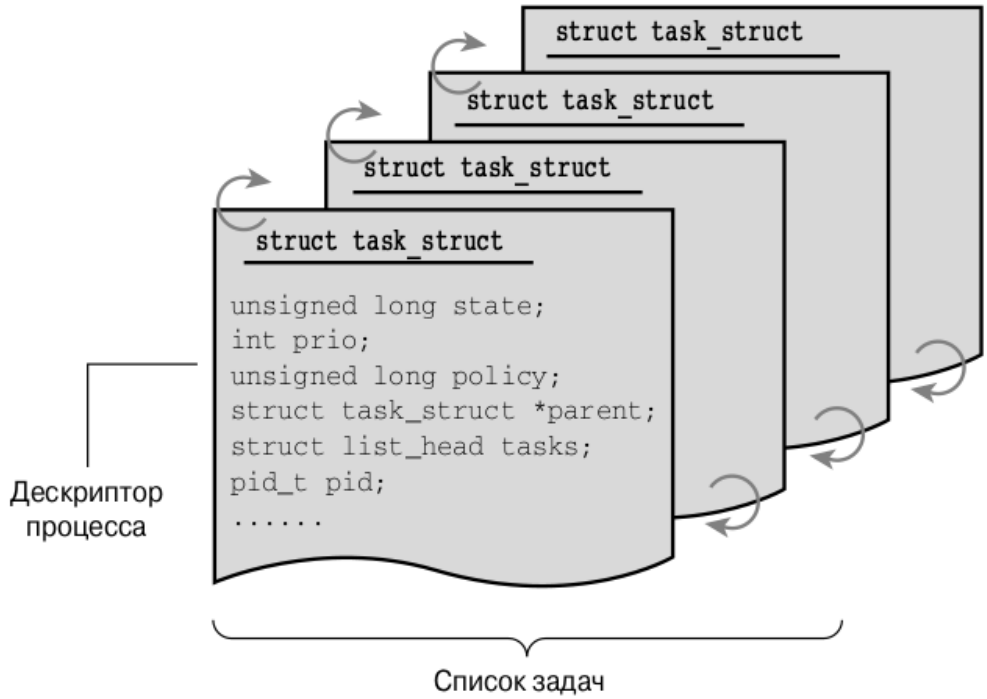

Рисунок 1.3 — Дескриптор процесса и список задач

Листинг 1.1 — Структура task_struct

```
1  struct task_struct {
2      void                *stack;
3      refcount_t          usage;
4      unsigned int            flags;
5      unsigned int            ptrace;
6      struct task_struct      *last_wakee;
7
8      ...
9
10     unsigned int            policy;
11
12     struct list_head        tasks;
13
14     struct mm_struct        *mm;
15     struct mm_struct        *active_mm;
16
17     /* Per-thread vma caching: */
18     struct vmacache         vmacache;
19
20     int             exit_state;
21     int             exit_code;
22     int             exit_signal;
23
24     pid_t               pid;
25     pid_t               tgid;
```

```
26
27
28      /* Real parent process: */
29      struct task_struct __rcu    *real_parent;
30
31      /* Recipient of SIGCHLD, wait4() reports: */
32      struct task_struct __rcu    *parent;
33
34      /*
35      * Children/sibling form the list of natural children:
36      */
37      struct list_head        children;
38      struct list_head        sibling;
39      struct task_struct      *group_leader;
40
41      /* Filesystem information: */
42      struct fs_struct        *fs;
43
44      /* Open file information: */
45      struct files_struct     *files;
46
47      /* Namespaces: */
48      struct nsproxy          *nsproxy;
49
50      /* Signal handlers: */
51      struct signal_struct        *signal;
52      struct sighand_struct       *sighand;
53      sigset_t            blocked;
54      sigset_t            real_blocked;
55
56      /* VM state: */
57      struct reclaim_state        *reclaim_state;
58
59      struct backing_dev_info     *backing_dev_info;
60
61      struct io_context       *io_context;
62
63      ...
64
65  };
```

1.2.2 Структура mm_struct.

Адресное пространство процесса представляется в ядре в виде структуры данных, которая называется дескриптором памяти (memory descriptor). В этой структуре содержится вся информация, относящаяся к адресному пространству процесса.

Дескриптор памяти представляется с помощью структуры mm_struct, которая определена в файле <linux/mm_types.h>. Указатель на данную структуру содерержится в поле mm структуры task_struct. Структура вместе с поясняющими комментариями по каждому полю приведена на листинге 1.2

Листинг 1.2 — Структура mm_struct

```
1  struct mm_struct {
2      struct vm_area_struct *mmap;
3      /* Список областей памяти */
4      struct rb_root mm_rb;
5      /* Красно–черное дерево областей памяти */
6      struct vm_area_struct *mmap_cache;
7      /* Последняя использованная область памяти */
8      unsigned long free_area_cache;
9      /* Первый незанятый участок адресного пространства */
10     pgd_t *pgd;
11     /* Глобальный каталог страниц */
12     atomic_t mm_users;
13     /* Счетчик использования адресного пространства */
14     atomic_t mm_count;
15     /* Основной счетчик использования */
16     int map_count;
17     /* Количество областей памяти */
18     struct rw_semaphore mmap_sem;
19     /* Семафор для областей памяти */
20     spinlock_t page_table_lock;
21     /* Спин–блокировка таблиц страниц */
22     struct list_head mmlist;
23     /* Список всех структур mm_struct */
24     unsigned long start_code;
25     /* Начальный адрес сегмента кода */
26     unsigned long end_code;
27     /* Конечный адрес сегмента кода */
28     unsigned long start_data;
29     /* Начальный адрес сегмента данных */
30     unsigned long end_data;
31     /* Конечный адрес сегмента данных */
32     unsigned long start_brk;
33     /* Начальный адрес сегмента "кучи" */
34     unsigned long brk;
35     /* Конечный адрес сегмента "кучи" */
36     unsigned long start_stack;
37     /* Начало стека процесса */
38     unsigned long arg_start;
39     /* Начальный адрес области аргументов */
40     unsigned long arg_end;
```

```
41      /* Конечный адрес области аргументов */
42      unsigned long env_start;
43      /* Начальный адрес области переменных среды */
44      unsigned long env_end;
45      /* Конечный адрес области переменных среды */
46      unsigned long rss;
47      /* rss - Количество распределенных физических страниц памяти */
48      unsigned long total_vm;
49      /* total_vm - Общее количество страниц памяти */
50      unsigned long locked_vm;
51      /* Количество заблокированных страниц памяти */
52      unsigned long saved_auxv[AT_VECTOR_SIZE];
53      /* Сохраненный вектор auxv */
54      cpumask_t cpu_vm_mask;
55      /* Маска отложенного переключения буфера TLB */
56      mm_context_t context;
57      /* Данные, специфичные для аппаратной платформы */
58      unsigned long  flags;
59      /* Флаги состояния */
60      int core_waiters;
61      /* количество потоков, ожидающих создания файла дампа */
62      struct core_state *core_state;
63      /* Поддержка дампа */
64      spinlock_t ioctx_lock;
65      /* Блокировка списка асинхронного ввода-вывода (AIO) */
66      struct hlist_head ioctx_list;
67      /* Список асинхронного ввода-вывода (AIO) */
68  };
```

В поле mm_users хранится количество процессов, в которых используется данное адресное пространство. Например, если одно и то же адресное пространство используется в двух потоках, значение поля mm_users равно 2.

В полях mmap и mm_rb хранятся ссылки на две различные структуры данных, содержащие одну и ту же информацию: информацию обо всех областях памяти в соответствующем адресном пространстве. В первой структуре эта информация хранится в виде связанного списка, а во второй — в виде красно-черного дерева. Поскольку красно-черное дерево — это разновидность двоичного дерева, то, как и для всех типов двоичных деревьев, количество операций поиска заданного элемента в нем подчиняется закону O(log(n)).

Все структуры mm_struct объединены в двухсвязный список с помощью полей mmlist.

1.2.3 Структура vm_area_struct

Области памяти (memory areas) представляются с помощью структуры vm_area_struct, которая определена в файле <linux/mm_types.h>.

Структура vm_area_struct используется для описания одной непрерывной области памяти в данном адресном пространстве. В ядре каждая область памяти считается уникальным объектом. Для каждой области памяти определены некоторые общие свойства, такие как права доступа и набор соответствующих операций. Таким образом, каждая структура VMA может представлять различный тип области памяти, например файлы, отображаемые в память, или стек пользовательского приложения. Структура vm_area_struct приведена на листинге 1.3.

Листинг 1.3 — Структура vm_area_struct

```
1  struct vm_area_struct {
2      struct mm_struct *vm_mm;
3      /* Соответствующая структура mm_struct */
4      unsigned long vm_start;/* Начало диапазона адресов (включительно) */
5      unsigned long vm_end; /* Конец диапазона адресов (исключая) */
6      struct vm_area_struct *vm_next; /* Список областей VMA */
7      pgprot_t vm_page_prot; /* Права доступа */
8      unsigned long vm_flags;/* Флаги */
9      struct rb_node vm_rb; /* Узел текущей области VMA в дереве */
10     union {
11         /* Связь с address_space->i_mmap или i_mmap_nonlinear */
12         struct {
13             struct list_head list;
14             void *parent;
15             struct vm_area_struct *head;
16         } vm_set;
17
18         struct prio_tree_node prio_tree_node;
19     } shared;
20     struct list_head anon_vma_node; /* Элемент анонимной области */
21     struct anon_vma *anon_vma; /* Объект анонимной VMA */
22     struct vm_operations_struct *vm_ops; /* Связанные операции */
23     unsigned long vm_pgoff; /* Смещение в файле */
24     struct file *vm_file; /* Отображенный файл (если есть) */
25     void *vm_private_data; /* Частные данные */
26 };
```

1.2.4 Взаимосвязь приведенных структур

Взаимосвязь приведенных структур продемонстрирована на рис. 1.4

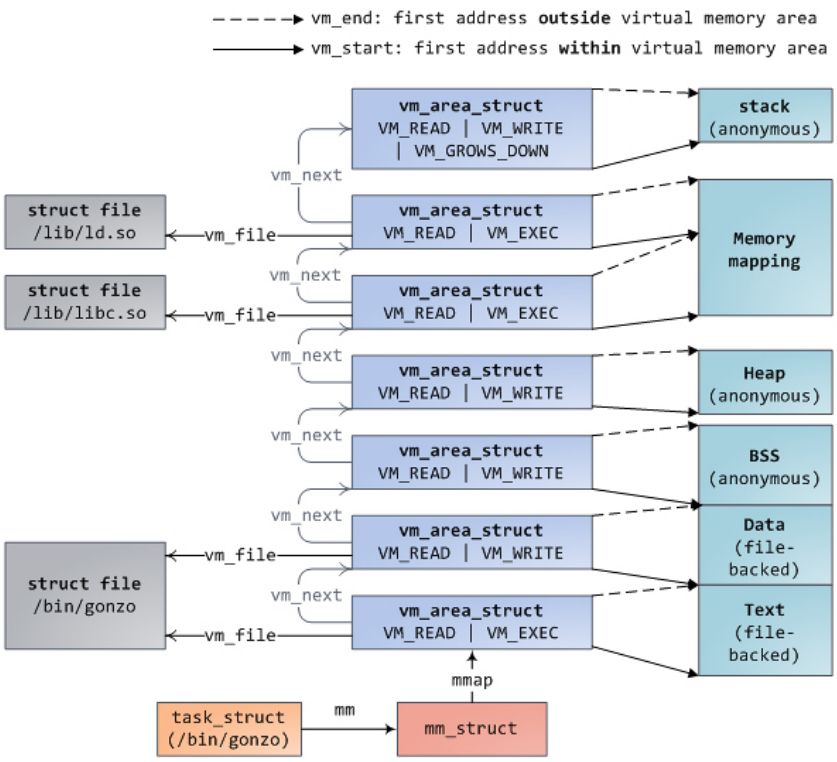

Рисунок 1.4 — Взаимосвязь приведенных структур

1.3 Прерывания

Прерывания делятся на:

— исключения (деление на ноль, переполнение стека), синхронные;

— системные вызовы (программные) - вызываются с помощью соответствующей команды из программы (int 21h), синхронные;

— аппаратные прерывания (прерывания от системного таймера, клавиатуры), асинхронные.

Прерывания делятся на 2 группы:

— быстрые;

— медленные.

Для того чтобы сократить время обработки медленных прерываний, они делятся на 2 части:

— top half, верхняя половина, запускается в результате получения процессором сигнала прерывания;

— bottom half, нижняя половина, отложенные вызовы.

Существует несколько способов реализации “нижней половины“ обработчика:

— softirq;

— тасклет (tasklet);

— очереди работ (workqueue).

1.3.1 Обработчики аппаратных прерываний

// TODO: total_vm это не физич у тебя написано физи где-то TODO: Написать про то, что я написала обработчика про то что из структур достаю информацию во время выпорлнения в очереди.

Обработчик прерывания должен выполнять минимальный объем действий и завершаться как можно быстрее. Обычно такой обработчик прерывания сохраняет данные, поступившие от внешнего устройства, в буфере ядра. Но для того чтобы обработать прерывания полностью, обработчик аппаратного прерывания должен инициализировать постановку в очередь на выполнение отложенное действие.

1.3.2 Очереди работ

Очереди работ являются обобщенным механизмом отложенного выполнения, в котором функция обработчика, реализующая соответствующие действия, может блокироваться.

struct workqueue_struct - описывает очередь работ.

Листинг 1.4 — Структура workqueue_struct

```
1 struct workqueue_struct {
2     struct list_head    pwqs;          /* WR: all pwqs of this wq */
3     struct list_head    list;          /* PR: list of all workqueues */
4     ...
5     struct pool_workqueue    *dfl_pwq;     /* PW: only for unbound wqs */
6     ...
7     struct pool_workqueue __percpu *cpu_pwqs; /* I: per-cpu pwqs */
8     ...
9     };
```

struct work_struct - описывает работу (обработчик нижней половины).

Листинг 1.5 — Структура workqueue_struct

```
1 struct work_struct {
2     atomic_long_t data;
3     struct list_head entry;
4     work_func_t func;
5     ...
6 };
```

Работа может инициализироваться 2-мя способами:

— статически;

— динамически.

При статической инициализации используется макрос:

Листинг 1.6 — статическая инициализация

```
1   DECLARE_WORK(name, void func)(void );
```

где: name – имя структуры work_struct, func – функция, которая вызывается из workqueue – обработчик нижней половины.

При динамической инициализации используются макросы:

Листинг 1.7 — динамическая инициализация

```
1   INIT_WORK(sruct work_struct *work, void func)(void),void *data);
```

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Это можно сделать несколькими способами. Во-первых, можно добавить работу (объект work) в очередь работ с помощью функции queue_work (которая назначает работу текущему процессору). Во-вторых, можно с помощью функции queue_work_on указать процессор, на котором будет выполняться обработчик.

1.4 Вывод

Были рассмотрены основополагающие материалы, которые в дальнейшем потребуются при реализации загружаемого модуля ядра.

2 Конструкторский раздел

2.1 Требования к программе

Необходимо реализовать загружаемый модуль ядра, который будет по нажатию на клавишу ESC выводить информацию о виртуальной памяти процесса, что позволит проанализировать данную информацию.

2.2 Анализируемая программа

В качестве анализируемой программы была выбрана программа, которая запускала n потоков. Каждый поток создавал свой собственный кольцевой односвязный список. Далее каждый поток пробегался по всему своему односвязному списку и обновлял значения (заполнял рандомными значениями). После обновления списка в конец добавлялся новый узел и приведенные выше операции повторялись вновь. На рис. 2.1 показан кольцевой односвязный список, использующийся в анализируемой программе.

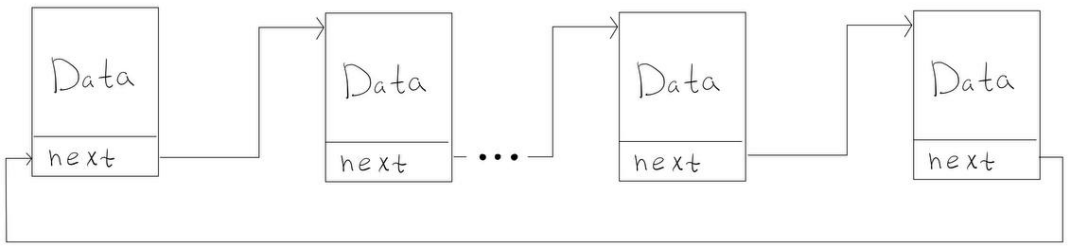

Рисунок 2.1 — Кольцевой односвязный список

На рис. 2.2 блок-схема алгоритма, который выполняет каждый поток.

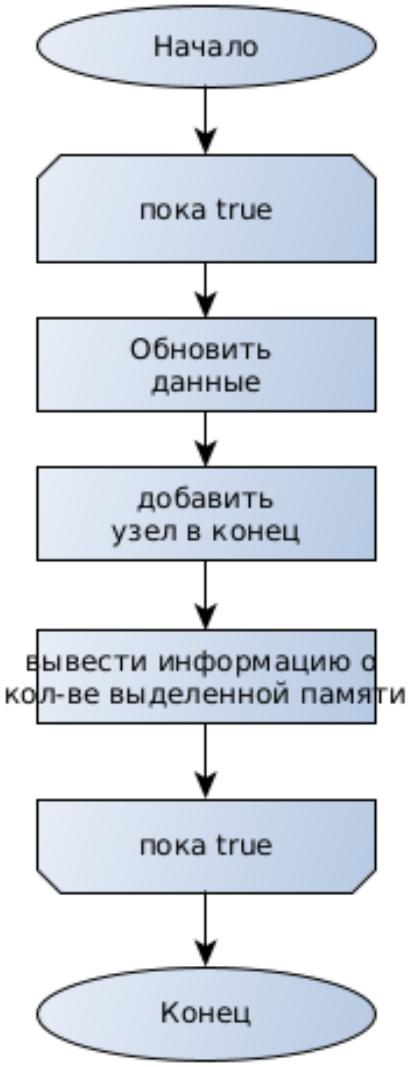

Рисунок 2.2 — Блок-схема алгоритма выполнения потока

2.3 Вывод

В данном разделе были рассмотрены требования к программе.

3 Технологический раздел

3.1 Выбор языка программирования и среды разработки

В данной курсовой работе использовался язык программирования - c [1].

В качестве среды разработки я использовала Visual Studio Code [2], т.к. считаю его достаточно удобным и легким. Visual Studio Code подходит не только для Windows [3], но и для Linux [4], это еще одна причина, по которой я выбрала VS code, т.к. у меня установлена ОС Ubuntu 18.04.4 [5].

3.2 Структура курсового проекта

Курсовой проект состоит из:

— md.c - загружаемого модуля ядра;

— my_mem_prog.c - анализируемой программы.

— additional_program.c - дополнительной программы, позволяющей выделять необходимое количество памяти, для дальнейшего анализа.

3.3 Листинг загружаемого модуля ядра

На листинге 3.1 приведен код загружаемого моду ядра.

Листинг 3.1 — Загружаемый модуль ядра

```
1  MODULE_LICENSE("GPL");
2  MODULE_AUTHOR("Alice");
3  MODULE_DESCRIPTION("Coursework BMSTU");
4
5  #define KBD_DATA_REG 0x60 /* I/O port for keyboard data */
6  #define KBD_SCANCODE_MASK 0x7f
7  #define KBD_STATUS_MASK 0x80
8
9
10 #define ANALYSIS_PROGRAM_NAME "my_mem_prog"
11
12 #define MONITORING_SCANCODE 1 // "[ESC]"
13
14 unsigned int my_irq = 1; // Прерывание от клавиатуры.
15
16 static struct workqueue_struct *my_wq; //очередь работ
17 struct task_struct *task = NULL;
18 char current_scancode;
19
20 bool find_user_task_struct(char* prog_name);
21 static void my_wq_function(struct work_struct *work);
22 irqreturn_t irq_handler(int irq, void *dev, struct pt_regs *regs);
23 void update_info_about_mem(struct mm_struct *info_about_mem);
```

```
24
25 DECLARE_WORK(my_work, my_wq_function);
26
27 unsigned long  total_vm_old;
28 unsigned long  locked_vm_old;
29 int  map_count_old;
30 unsigned long  all_brk_old;
31
32 void info_about_mm(void)
33 {
34     struct mm_struct *info_about_mem;
35     bool is_new_data = false;
36
37     unsigned long  total_vm_current;
38     unsigned long  locked_vm_current;
39     int map_count_current;
40     unsigned long  all_brk_current;
41
42     if (find_user_task_struct(ANALYSIS_PROGRAM_NAME) == false)
43     {
44         printk(KERN_INFO "+Module: find_user_task_struct is false");
45         return;
46     }
47
48     info_about_mem = task->mm;
49
50     total_vm_current = info_about_mem->total_vm;
51     locked_vm_current = info_about_mem->locked_vm;
52     map_count_current = info_about_mem->map_count;
53     all_brk_current = info_about_mem->brk - info_about_mem->start_brk;
54
55     // update_info_about_mem(info_about_mem);
56
57     if (total_vm_old > total_vm_current)
58     {
59         total_vm_old = total_vm_current;
60         return;
61     }
62
63     if (all_brk_old > all_brk_current)
64     {
65         all_brk_old = all_brk_current;
66         return;
67     }
68
69
70     if (total_vm_current != total_vm_old)
```

```
71      {
72          printk(KERN_INFO "+Module: Общее количество страниц памяти: было:
                %lu; стало:%lu; разница:%lu", total_vm_old, total_vm_current,
                total_vm_current - total_vm_old);
73          total_vm_old = total_vm_current;
74          is_new_data = true;
75      }
76
77      if (all_brk_current != all_brk_old)
78      {
79          printk(KERN_INFO "+Module: Используется сегментом кучи: было: %lu;
                стало:%lu; разница:%lu", all_brk_old, all_brk_current,
                all_brk_current - all_brk_old);
80          all_brk_old = all_brk_current;
81          is_new_data = true;
82      }
83
84      if (!is_new_data)
85      {
86          printk(KERN_INFO "+Module: Нет изменений");
87      }
88  }
89
90  static void my_wq_function(struct work_struct *work) // вызываемая функция
91  {
92      int scan_normal;
93
94      if (!(current_scancode & KBD_STATUS_MASK))
95      {
96          scan_normal = current_scancode & KBD_SCANCODE_MASK;
97
98          if (scan_normal == MONITORING_SCANCODE)
99          {
100             info_about_mm();
101         }
102     }
103 }
104
105 irqreturn_t irq_handler(int irq, void *dev, struct pt_regs *regs)
106 {
107     if (irq == my_irq)
108     {
109         // Получаем скан-код нажатой клавиши.
110         current_scancode = inb(KBD_DATA_REG);
111         queue_work(my_wq, &my_work);
112
113         return IRQ_HANDLED; // прерывание обработано
```

```
114     }
115     else
116         return IRQ_NONE; // прерывание не обработано
117 }
118
119 bool find_user_task_struct(char* prog_name)
120 {
121     struct task_struct *current_task = &init_task;
122
123     do {
124         if (!strcmp(current_task->comm, prog_name))
125         {
126             task = current_task;
127             return true;
128         }
129     } while ((current_task = next_task(current_task)) != &init_task);
130
131     return false;
132 }
133
134 void update_info_about_mem(struct mm_struct *info_about_mem)
135 {
136     atomic_t mm_users;
137     int counter;
138
139     mm_users = info_about_mem->mm_users; /* Счетчик использования адресного
            пространства */
140     counter = mm_users.counter;
141
142     total_vm_old = info_about_mem->total_vm;
143     locked_vm_old = info_about_mem->locked_vm;
144     map_count_old = info_about_mem->map_count;
145     all_brk_old = info_about_mem->brk - info_about_mem->start_brk;
146
147     printk(KERN_INFO "+Module: Количество процессов, в которых используется
            данное адресное пространство: %d", counter);
148
149     printk(KERN_INFO "+Module: Общее количество страниц памяти = %lu ",
            total_vm_old);
150     printk(KERN_INFO "+Module: Количество заблокированных страниц памяти =
            %lu ", locked_vm_old);
151     printk(KERN_INFO "+Module: Количество областей памяти: %d",
            map_count_old);
152
153     printk(KERN_INFO "+Module: Используется сегментом кучи: %lu",
            all_brk_old);
```

```
154     printk(KERN_INFO "+Module: Используется сегментом кода: %lu",
            info_about_mem->end_code - info_about_mem->start_code);
155     printk(KERN_INFO "+Module: Используется сегментом данных: %lu",
            info_about_mem->end_data - info_about_mem->start_data);
156 }
157
158
159 void first_proc(void)
160 {
161     struct mm_struct *info_about_mem;
162
163
164     if (find_user_task_struct(ANALYSIS_PROGRAM_NAME) == false)
165     {
166         printk(KERN_INFO "+Module: find_user_task_struct is false");
167         return;
168     }
169
170     info_about_mem = task->mm;
171
172     update_info_about_mem(info_about_mem);
173 }
174
175 static int __init md_init(void)
176 {
177     // регистрация обработчика прерывания
178     if (request_irq(
179             my_irq,                         /* номер irq */
180             (irq_handler_t)irq_handler,     /* наш обработчик */
181             IRQF_SHARED,                    /* линия может быть раздедена, IRQ
182                                             (разрешено совместное использование
                                                )*/
183             "my_irq_handler",               /* имя устройства (можно потом посм
                отреть в /proc/interrupts)*/
184             (void *)(irq_handler)))         /* Последний параметр
                (идентификатор устройства) irq_handler нужен
185                                             для того, чтобы можно отключить с п
                                                омощью free_irq  */
186     {
187         printk(" + Error request_irq");
188         return -1;
189     }
190
191     my_wq = create_workqueue("my_queue"); //создание очереди работ
192     if (my_wq)
193     {
194         printk(KERN_INFO "Module: workqueue created!\n");
```

```
195     }
196     else
197     {
198         free_irq(my_irq, irq_handler); // Отключение обработчика прерывания
                .
199         printk(KERN_INFO "Module: error create_workqueue()!\n");
200         return -1;
201     }
202
203     first_proc();
204     // find_user_task_struct(ANALYSIS_PROGRAM_NAME);
205
206     printk(KERN_INFO "Module: module loaded!\n");
207     return 0;
208 }
209
210 static void __exit md_exit(void)
211 {
212     // Принудительно завершаем все работы в очереди.
213     // Вызывающий блок блокируется до тех пор, пока операция не будет завер
        шена.
214     flush_workqueue(my_wq);
215     destroy_workqueue(my_wq);
216
217     // my_irq - номер прерывания.
218     // irq_handler - идентификатор устройства.
219     free_irq(my_irq, irq_handler); // Отключение обработчика прерывания.
220
221     printk(KERN_INFO "Module: unloaded!\n");
222 }
223
224 module_init(md_init);
225 module_exit(md_exit);
```

3.4 Анализируемая программа

На листинге 3.2 приведен код анализируемой программы.

Листинг 3.2 — Анализируемая программа

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <pthread.h>
7
```

```
8  #define SUCCESS 0
9
10 #define ERROR_CREATE_THREAD -11
11 #define ERROR_JOIN_THREAD   -12
12
13 #define VALUE_SIZE 64
14 #define PTHREAD_COUNT 3
15 #define SLEEP_TIME 3
16
17 #define GET_RAND_NUMBER(min, max) (rand() % (max - min + 1) + min)
18
19 typedef struct Node {
20     int *value; // VALUE_SIZE
21     struct Node *next; // 8 byte
22 } Node;
23
24 Node *create()
25 {
26     Node *node = (Node*)malloc(sizeof(Node));
27     node->value = (int*)malloc(VALUE_SIZE * sizeof(int)); // VALUE_SIZE *
           4 (npu 64 == 256)
28     node->next = NULL;
29     return node;
30 }
31
32 void add_to_end(Node *node)
33 {
34     Node *new_node = create();
35     node->next = new_node;
36 }
37
38 void output_data(int *data)
39 {
40     for (int i = 0; i < VALUE_SIZE; i++)
41     {
42         printf("%d ", data[i]);
43     }
44     printf("\n");
45 }
46
47 void output(Node *node)
48 {
49     while(node != NULL)
50     {
51         output_data(node->value);
52         node = node->next;
53     }
```

```
54      printf("\n\n");
55  }
56
57  void generate_random_values(Node *node)
58  {
59      for (int i = 0; i < VALUE_SIZE; i++)
60      {
61          node->value[i] = GET_RAND_NUMBER(0, 100);
62      }
63  }
64
65  void update_list(Node *node)
66  {
67      while(node != NULL)
68      {
69          generate_random_values(node);
70          node = node->next;
71      }
72  }
73
74  int msleep(long msec)
75  {
76      struct timespec ts;
77      int res;
78
79      if (msec < 0)
80      {
81          errno = EINVAL;
82          return -1;
83      }
84
85      ts.tv_sec = msec / 1000;
86      ts.tv_nsec = (msec % 1000) * 1000000;
87
88      do {
89          res = nanosleep(&ts, &ts);
90      } while (res && errno == EINTR);
91
92      return res;
93  }
94
95  void process(char* name)
96  {
97      Node *first = create();
98      Node *current_node = first;
99
100     long long int i = 0;
```

```
101     while (1)
102     {
103         update_list(first);
104 
105         add_to_end(current_node);
106 
107         current_node = current_node->next;
108 
109         i++;
110         if (!(i % 16))
111         {
112             printf("1 kilobytes\n");
113         }
114 
115         long long int byte = (256 + 8) * i;
116         printf(" %s byte = %lld kilobyte = %lld ",name, byte, byte / 1024);
117         printf("pages = %lld \n", byte / 1024 / 4);
118         msleep(SLEEP_TIME);
119     }
120 }
121 
122 void* do_pthread(void *args)
123 {
124     process((char*)args);
125     return SUCCESS;
126 }
127 
128 int main()
129 {
130     printf("Start program");
131     srand(time(NULL));
132     setbuf(stdout, NULL);
133 
134     pthread_t threads[PTHREAD_COUNT];
135     char* names[5] = {"1", "2", "3", "4", "5"};
136 
137     int status;
138     int status_addr;
139 
140     for (int i = 0; i < PTHREAD_COUNT; i++)
141     {
142         status = pthread_create(&threads[i], NULL, do_pthread, names[i]);
143         if (status != 0) {
144             printf("main error: can't create thread, status = %d\n",
                    status);
145             exit(ERROR_CREATE_THREAD);
146         }
```

```
147     }
148
149     do_pthread("Main pthread");
150 }
```

3.5 Вспомогательная программа

На листинге 3.3 приведена вспомогательная программа. Она предоставляет возможность выделения памяти, размер которой равен 1024 * 4 * n байт (n страниц). Где n - число, которое вводит пользователь.

Листинг 3.3 — Вспомогательная программа

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define PAGE_SIZE 4 * 1024 // 4 kilobyte
6  #define OK 0
7
8  void create(int page_count)
9  {
10     void* mem = (void*)malloc(PAGE_SIZE * page_count);
11     printf("malloc (%d pages)\n", page_count);
12 }
13
14 void process(void)
15 {
16     int page_count = 1;
17     while (page_count)
18     {
19         create(page_count);
20         printf("Input page count (0 to exit) : ");
21         scanf("%d", &page_count);
22     }
23 }
24
25 int main()
26 {
27     srand(time(NULL));
28     setbuf(stdout, NULL);
29     process();
30     return OK;
31 }
```

3.6 Вывод

В данном разделе был выбран язык программирования и среда разработки. А также представлены листинги.

4 Экспериментальная часть

4.1 Анализ увелечения количества страниц от количества потоков

На рис. 4.1 демонстрируется увеличение количества страниц от количества потоков (при первом запуске программы).

При одном главном потоке выделяется 1675 страниц.

При двух потоках выделяется 20108 страниц (примерно в 12 раз больше, чем при одном потоке).

Далее при увеличении количества потоков количество страниц увеличивается на 18433 страницы.

При единственном главном потоке (когда не создаются дочерние потоки) выделяется определенное количество страниц. Далее при первом вызове функции pthread_create выделяется некоторое количество страниц. Все последующие вызовы функции pthread_create будут запрашивать определенное одинаковое количество страниц.

```
[ 1308.202996] +Module: Количество процессов, в которых используется данное адресное пространство: 1
[ 1308.202999] +Module: Общее количество страниц памяти = 1675
[ 1308.203000] +Module: Количество заблокированных страниц памяти = 0
[ 1308.203002] +Module: Количество областей памяти: 22
[ 1308.203004] +Module: Используется сегментом кучи: 135168
[ 1308.203006] +Module: Используется сегментом кода: 5032
[ 1308.203007] +Module: Используется сегментом данных: 720
[ 1308.203008] +Module: Нет изменений
[ 1327.614826] +Module: Количество процессов, в которых используется данное адресное пространство: 2
[ 1327.614831] +Module: Общее количество страниц памяти = 20108
[ 1327.614833] +Module: Количество заблокированных страниц памяти = 0
[ 1327.614835] +Module: Количество областей памяти: 26
[ 1327.614838] +Module: Используется сегментом кучи: 135168
[ 1327.614840] +Module: Используется сегментом кода: 5032
[ 1327.614842] +Module: Используется сегментом данных: 720
[ 1327.614843] +Module: Нет изменений
[ 1352.808282] +Module: Количество процессов, в которых используется данное адресное пространство: 3
[ 1352.808288] +Module: Общее количество страниц памяти = 38541
[ 1352.808289] +Module: Количество заблокированных страниц памяти = 0
[ 1352.808291] +Module: Количество областей памяти: 30
[ 1352.808293] +Module: Используется сегментом кучи: 135168
[ 1352.808294] +Module: Используется сегментом кода: 5032
[ 1352.808296] +Module: Используется сегментом данных: 720
[ 1352.808297] +Module: Нет изменений
[ 1367.214412] +Module: Количество процессов, в которых используется данное адресное пространство: 4
[ 1367.214413] +Module: Общее количество страниц памяти = 56974
[ 1367.214414] +Module: Количество заблокированных страниц памяти = 0
[ 1367.214415] +Module: Количество областей памяти: 34
[ 1367.214416] +Module: Используется сегментом кучи: 135168
[ 1367.214416] +Module: Используется сегментом кода: 5032
[ 1367.214417] +Module: Используется сегментом данных: 720
[ 1367.214418] +Module: Нет изменений
[ 1415.570551] +Module: Количество процессов, в которых используется данное адресное пространство: 5
[ 1415.570555] +Module: Общее количество страниц памяти = 75407
[ 1415.570558] +Module: Количество заблокированных страниц памяти = 0
[ 1415.570560] +Module: Количество областей памяти: 38
[ 1415.570562] +Module: Используется сегментом кучи: 135168
[ 1415.570563] +Module: Используется сегментом кода: 5032
[ 1415.570565] +Module: Используется сегментом данных: 720
[ 1415.570567] +Module: Нет изменений
[ 1427.441872] +Module: Количество процессов, в которых используется данное адресное пространство: 6
[ 1427.441876] +Module: Общее количество страниц памяти = 93840
[ 1427.441877] +Module: Количество заблокированных страниц памяти = 0
[ 1427.441879] +Module: Количество областей памяти: 42
[ 1427.441881] +Module: Используется сегментом кучи: 135168
[ 1427.441882] +Module: Используется сегментом кода: 5032
[ 1427.441884] +Module: Используется сегментом данных: 720
```

Рисунок 4.1 — Демонстрация увелечения количества страниц от количества потоков

4.2 Анализ увелечения количества страниц программы, описанной выше

На рис. 4.2 демонстрируется увеличение количества страниц при анализе описанной выше программы (при одном потоке).

Видно, что программе выделяется 33 страницы. Так же видно, что увеличивается размер кучи. Он увеличивается на 135168, что равно 33 * 4 * 1024, т.е. все 33 страницы выделяются под кучу.

```
[ 2971.495137] +Module: Общее количество страниц памяти: было: 1675; стало:1708; разница:33
[ 2971.495142] +Module: Используется сегментом кучи: было: 135168; стало:270336; разница:135168
[ 2971.531250] +Module: Нет изменений
[ 2971.567333] +Module: Нет изменений
[ 2976.488595] +Module: Общее количество страниц памяти: было: 1708; стало:1741; разница:33
[ 2976.488598] +Module: Используется сегментом кучи: было: 270336; стало:405504; разница:135168
[ 2981.306206] +Module: Общее количество страниц памяти: было: 1741; стало:1774; разница:33
[ 2981.306209] +Module: Используется сегментом кучи: было: 405504; стало:540672; разница:135168
[ 2986.496115] +Module: Общее количество страниц памяти: было: 1774; стало:1840; разница:66
[ 2986.496118] +Module: Используется сегментом кучи: было: 540672; стало:811008; разница:270336
[ 2988.708524] +Module: Нет изменений
[ 2989.682809] +Module: Нет изменений
[ 2991.051646] +Module: Общее количество страниц памяти: было: 1840; стало:1873; разница:33
[ 2991.051648] +Module: Используется сегментом кучи: было: 811008; стало:946176; разница:135168
[ 2992.763583] +Module: Нет изменений
[ 2993.558834] +Module: Нет изменений
[ 2994.248927] +Module: Общее количество страниц памяти: было: 1873; стало:1906; разница:33
```

Рисунок 4.2 — Демонстрация увеличение количества страниц при анализе описанной выше программы

4.3 Анализ увелечения количества страниц программы, описанной выше при увеличении количества потоков

На рис. 4.3 демонстрируется увеличение количества страниц при анализе описанной выше программы при увеличении количества потоков от 1 до 4.

Видно, что независимо, от изначального количества потоков программе выделяется 33 страницы.

```
[ 4684.383813] +Module: Нет изменений
[ 4684.884025] +Module: Общее количество страниц памяти: было: 1708; стало:1741; разница:33
[ 4684.884028] +Module: Используется сегментом кучи: было: 270336; стало:405504; разница:135168
[ 4685.454274] +Module: Нет изменений
[ 4687.002834] +Module: Нет изменений
[ 4695.516575] +Module: Общее количество страниц памяти: было: 1741; стало:20108; разница:18367
[ 4696.168996] +Module: Нет изменений
[ 4697.022339] +Module: Нет изменений
[ 4697.549551] +Module: Нет изменений
[ 4698.012082] +Module: Нет изменений
[ 4698.539207] +Module: Нет изменений
[ 4699.115171] +Module: Нет изменений
[ 4699.761112] +Module: Общее количество страниц памяти: было: 20108; стало:20141; разница:33
[ 4699.761112] +Module: Используется сегментом кучи: было: 135168; стало:270336; разница:135168
[ 4700.386238] +Module: Нет изменений
[ 4701.017479] +Module: Нет изменений
[ 4725.082782] +Module: Общее количество страниц памяти: было: 20141; стало:38541; разница:18400
[ 4725.555343] +Module: Нет изменений
[ 4726.193766] +Module: Нет изменений
[ 4729.928186] +Module: Общее количество страниц памяти: было: 38541; стало:38574; разница:33
[ 4729.928187] +Module: Используется сегментом кучи: было: 135168; стало:270336; разница:135168
[ 4733.672593] +Module: Нет изменений
[ 4735.734394] +Module: Общее количество страниц памяти: было: 38574; стало:38607; разница:33
[ 4735.734396] +Module: Используется сегментом кучи: было: 270336; стало:405504; разница:135168
[ 4748.014565] +Module: Общее количество страниц памяти: было: 38607; стало:56974; разница:18367
[ 4752.586612] +Module: Общее количество страниц памяти: было: 56974; стало:57007; разница:33
[ 4752.586613] +Module: Используется сегментом кучи: было: 135168; стало:270336; разница:135168
[ 4759.885776] +Module: Нет изменений
[ 4761.195186] +Module: Общее количество страниц памяти: было: 57007; стало:57040; разница:33
[ 4761.195188] +Module: Используется сегментом кучи: было: 270336; стало:405504; разница:135168
```

Рисунок 4.3 — Демонстрация увеличение количества страниц при анализе описанной выше программы при увеличение кол-ва потоков

4.4 Анализ вспомогательной программы

На рис. 4.4 запрашивается последовательно некоторое количество памяти, размер которой в сумме дает 33 * 4 * 1024 байта, что равно 33 страницам. Когда запрашиваемое количество памяти превышает 33 страницы, то общее количество страниц увеличивается на 33 страницы см. рисунок 4.5.

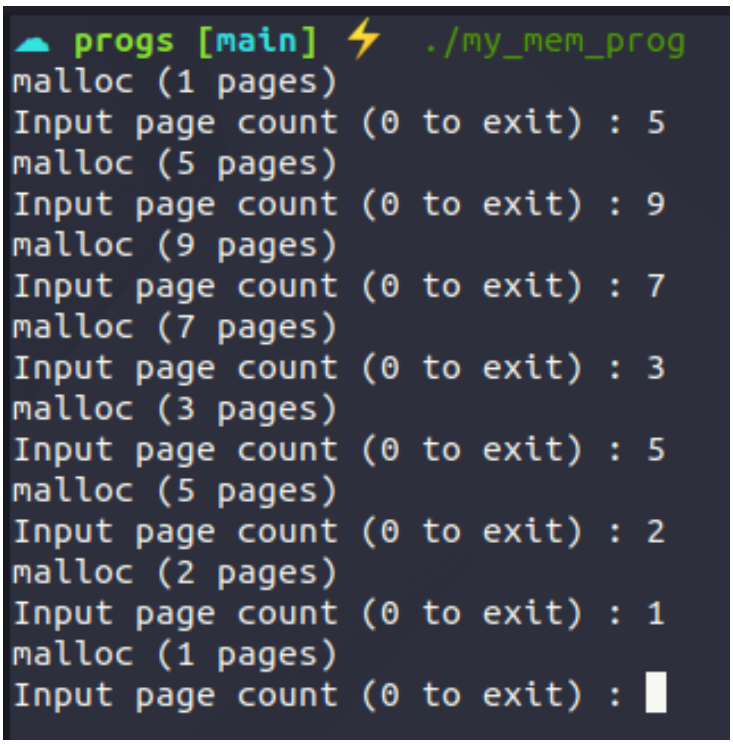

Рисунок 4.4 — Последовательный запрос малого количества страниц

```
+Module: Общее количество страниц памяти: было: 1131; стало:1164; разница:33
+Module: Используется сегментом кучи: было: 135168; стало:270336; разница:135168
+Module: Нет изменений
```

Рисунок 4.5 — Результат при последовательном запросе малого количества страниц

На рисунках 4.6 - 4.11 показано, что при запросе памяти, размер которой более 33 * 4 * 1024 байта, что равно 33 страницам, общее количество страниц увеличивается на количество запрашиваемых страниц + 1.

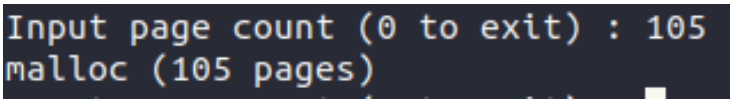

Рисунок 4.6 — Запрос 105 страниц

```
Нет изменений
Общее количество страниц памяти: было: 1257; стало:1363; разница:106
```

Рисунок 4.7 — Результат при запросе 105 страниц

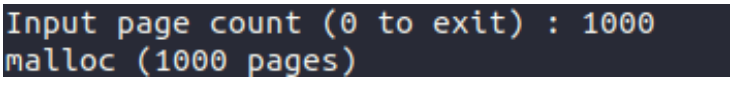

Рисунок 4.8 — Запрос 1000 страниц

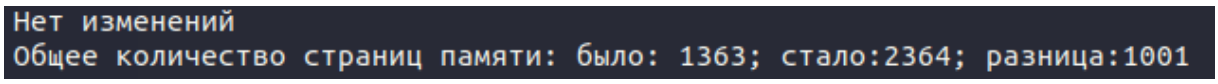

Рисунок 4.9 — Результат при запросе 1000 страниц

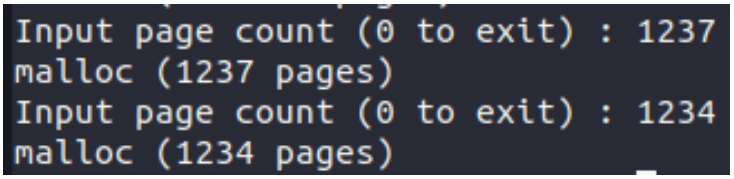

Рисунок 4.10 — Запрос 1237 и 1234 страниц

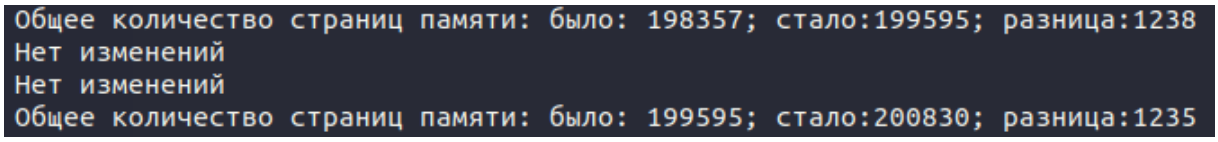

Рисунок 4.11 — Результат при запросе 1237 и 1234 страниц

4.5 Вывод

В данном разделе был приведен анализ общего количества выделенных страниц в зависимости от запрашиваемого количества памяти. Была проанализирована описанная выше программа.

Заключение

В рамках выполнения работы решены следующие задачи.

а) Рассмотрены основополагающие материалы, которые в дальнейшем потребовались при реализации загружаемого модуля ядра.

б) Рассмотрены требования к программе.

в) Был выбран язык программирования и среда разработки, а также представлены листинги.

г) Был написан загружаемый модуль ядра.

д) Была написана дополнительная анализируемая программа.

е) Проведен анализ дополнительной анализируемой программы.

ж) Была написана вспомогательная программа.

з) Проведен анализ количества выделяемых страниц в зависимости от количества выделяемой памяти.

Список использованных источников

1. *Керниган Брайан У., Ритчи Деннис М.* Язык программирования C / Ритчи Деннис М. Керниган Брайан У. — Вильямс, 2019. — P. 288.

2. Visual Studio Code. — Microsoft, 2005. `https://code.visualstudio.com/`.

3. Windows. — Microsoft, 1985. `https://www.microsoft.com/ru-ru/windows`.

4. Linux. — 1991. `https://www.linux.org.ru/`.

5. Ubuntu 18.04. — 2018. `https://releases.ubuntu.com/18.04/`.