1.

Содержание

Введение . . . 3
1 Аналитический раздел . . . 4
1.1 Адресное пространство процесса . . . 4
1.1.1 Виртуальная память . . . 4
1.1.2 Таблица страниц . . . 4
1.2 Структуры ядра . . . 6
1.2.1 Структура task_struct . . . 6
1.2.2 Структура mm_struct. . . . 8
1.2.3 Структура vm_area_struct . . . 11
1.2.4 Взаимосвязь приведенных структур . . . 11
1.3 Прерывания . . . 12
1.3.1 Обработчики аппаратных прерываний . . . 13
1.3.2 Очереди работ . . . 13
1.4 Вывод . . . 14
2 Констукторский раздел . . . 15
2.1 Требования к программе . . . 15
2.2 Анализируемая программа . . . 15
2.3 Вывод . . . 15
3 Технологический раздел . . . 17
3.1 Выбор языка программирования и среды разработки . . . 17
3.2 Требования к программному обеспечению . . . 17
3.3 Программное обеспечение . . . 17
3.4 Анализируемая программа . . . 17
3.5 Вывод . . . 21
4 Экспериментальная часть . . . 22
4.1 Временные характеристики . . . 22
4.2 Сравнительный анализ алгоритмов . . . 22
4.3 Вывод . . . 22
Заключение . . . 23
Список использованных источников . . . 24

Введение

Адресное пространство процесса состоит из виртуальной памяти, адресуемой процессом, и диапазона адресов в этой виртуальной памяти, которые разрешено использовать процессу. Данная курсовая работа предоставит информацию, которая не доступна в режиме пользователя.

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

1 Аналитический раздел

1.1 Адресное пространство процесса

Ядро управляет памятью пользовательских программ. Эта память называется адресным пространством процесса (process address space) и выделяется операционной системой каждому пользовательскому процессу. Операционная система Linux является системой с поддержкой виртуальной памяти, т.е. в ней выполняется виртуализация ресурсов памяти среди всех процессов в системе. Для каждого процесса создается иллюзия того, что он один использует всю физическую память в системе. Адресное пространство одного процесса может значительно превышать объем физической памяти компьютера.

1.1.1 Виртуальная память

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.

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 табли-

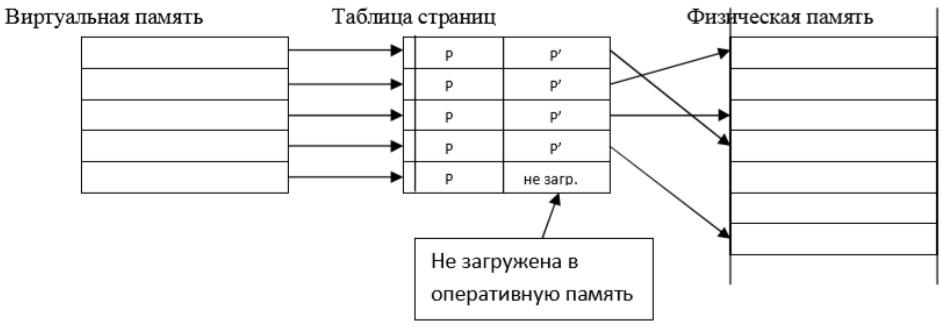

Рисунок 1.1 — Отображение виртуальной памяти на физическую с помощью таблицы страниц

цы 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.

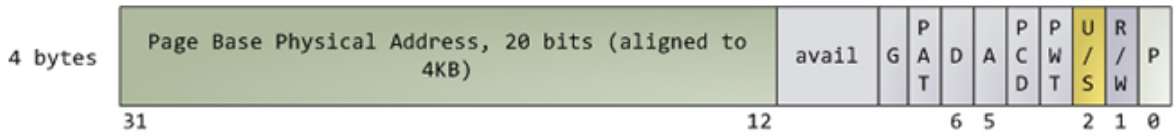

Рисунок 1.2 — 4-байтовая запись pgd

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 менедже-

ра 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.

Листинг 1.1 — Структура task_struct

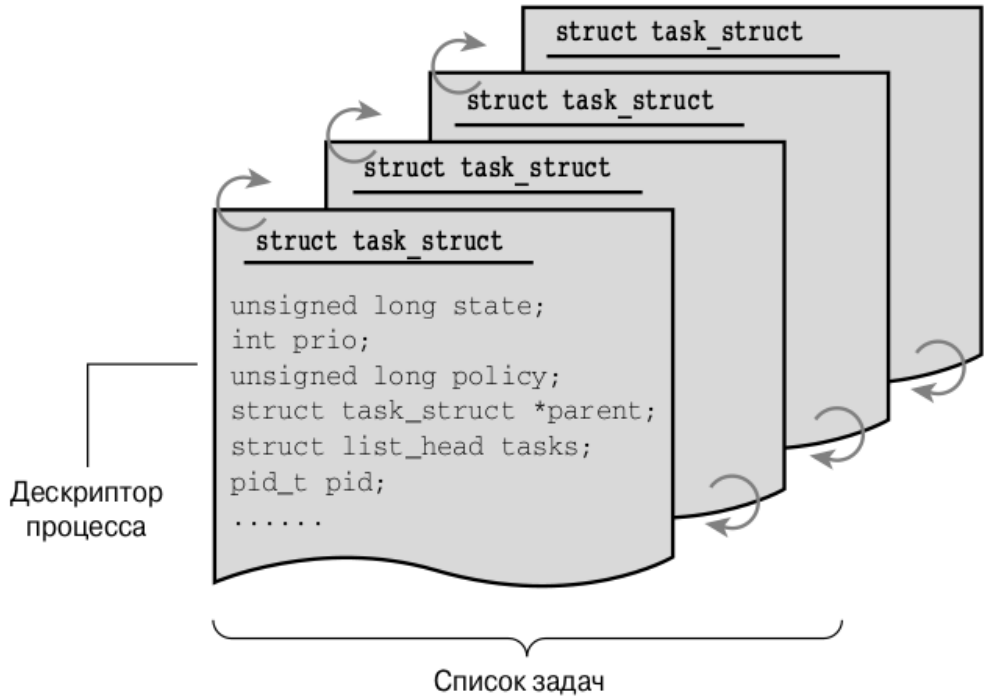

Рисунок 1.3 — Дескриптор процесса и список задач

```
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 опре-

делена 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      /* Количество распределенных физических страниц памяти */
48      unsigned long total_vm;
49      /* 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

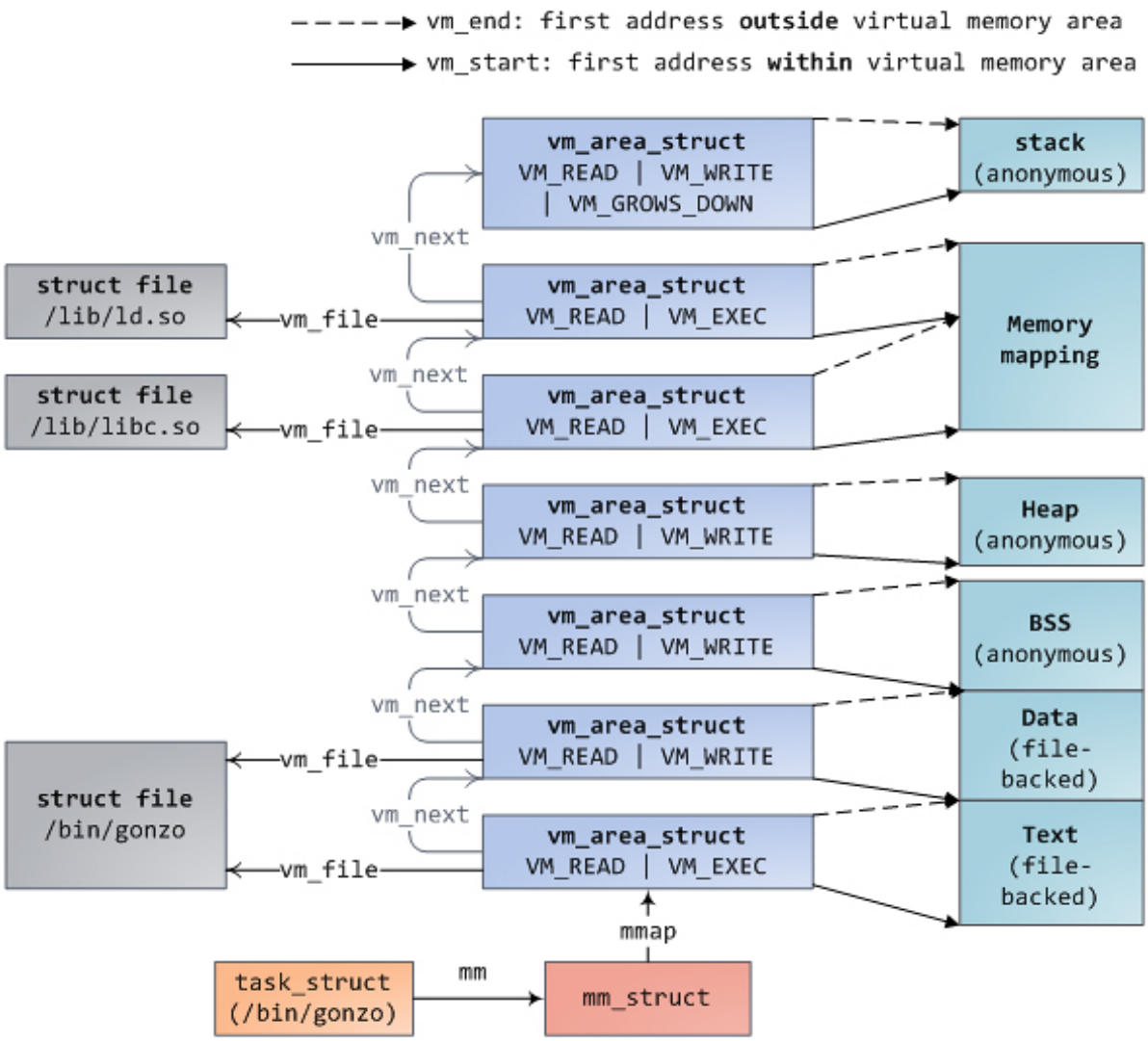

Рисунок 1.4 — Взаимосвязь приведенных структур

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

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 Констукторский раздел

2.1 Требования к программе

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.

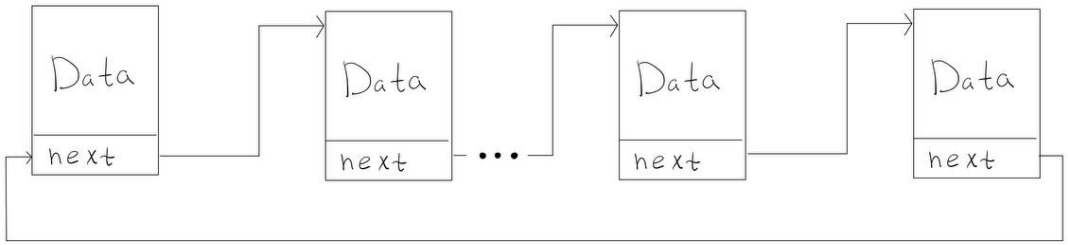

Рисунок 2.1 — Кольцевой односвязный список

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

2.3 Вывод

В данном разделе было рассмотрено ...

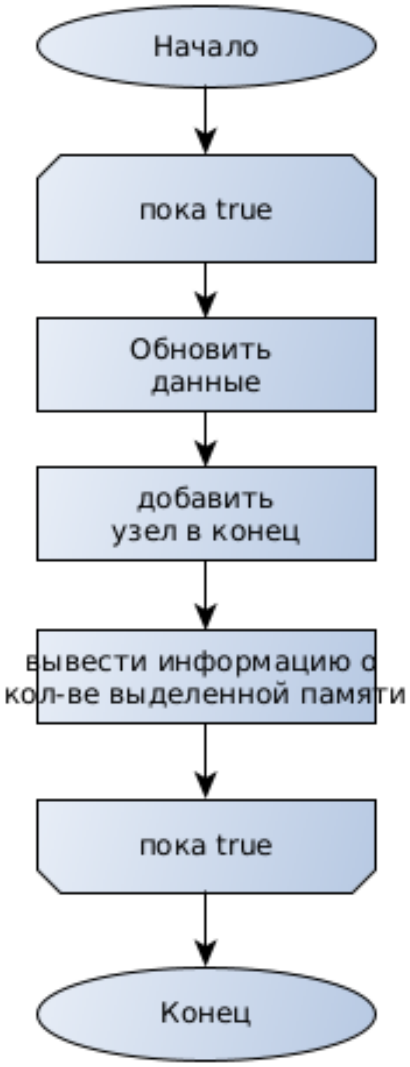

Рисунок 2.2 — Блок-схема алгоритма выполнения потока

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 Требования к программному обеспечению

Входными данными являются

3.3 Программное обеспечение

На листинге 3.1 приведен код загружаемого моду ядра.

Листинг 3.1 — Загружаемый модуль ядра

На выходе

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 VALUE_SIZE 64
9  #define OK 0
10 #define GET_RAND_NUMBER(min, max) (rand() % (max - min + 1) + min)
11 #define ERROR_CREATE_THREAD -11
12 #define ERROR_JOIN_THREAD   -12
13 #define SUCCESS          0
14 #define PTHREAD_COUNT 2
15
16 typedef struct Node {
17     int *value; // VALUE_SIZE
18     struct Node *next; // 8 byte
19 } Node;
20
```

```
21 Node *create()
22 {
23     Node *node = (Node*)malloc(sizeof(Node));
24     node->value = (int*)malloc(VALUE_SIZE * sizeof(int)); // VALUE_SIZE *
       4 (npu 64 == 256)
25     node->next = NULL;
26     return node;
27 }
28
29 void add_to_end(Node *node)
30 {
31     Node *new_node = create();
32     node->next = new_node;
33 }
34
35 void output_data(int *data)
36 {
37     for (int i = 0; i < VALUE_SIZE; i++)
38     {
39         printf("%d ", data[i]);
40     }
41     printf("\n");
42 }
43
44 void output(Node *node)
45 {
46     while(node != NULL)
47     {
48         output_data(node->value);
49         node = node->next;
50     }
51     printf("\n\n");
52 }
53
54 void generate_random_values(Node *node)
55 {
56     for (int i = 0; i < VALUE_SIZE; i++)
57     {
58         node->value[i] = GET_RAND_NUMBER(0, 100);
59     }
60 }
61
62 void update_list(Node *node)
63 {
64     while(node != NULL)
65     {
66         generate_random_values(node);
```

```
67          node = node->next;
68      }
69  }
70
71  int msleep(long msec)
72  {
73      struct timespec ts;
74      int res;
75
76      if (msec < 0)
77      {
78          errno = EINVAL;
79          return -1;
80      }
81
82      ts.tv_sec = msec / 1000;
83      ts.tv_nsec = (msec % 1000) * 1000000;
84
85      do {
86          res = nanosleep(&ts, &ts);
87      } while (res && errno == EINTR);
88
89      return res;
90  }
91
92  void process(char* name)
93  {
94      Node *first = create();
95      Node *current_node = first;
96
97      long long int i = 0;
98      while (1)
99      {
100         update_list(first);
101
102         add_to_end(current_node);
103
104         current_node = current_node->next;
105
106         i++;
107         if (!(i % 16))
108         {
109             printf("1 kilobytes\n");
110         }
111
112         long long int byte = (256 + 8) * i;
113         printf(" %s byte = %lld kilobyte = %lld ",name, byte, byte / 1024);
```

```
114             printf("pages = %lld \n", byte / 1024 / 4);
115             msleep(100);
116         }
117     }
118
119     void* do_pthread(void *args)
120     {
121         process((char*)args);
122         return SUCCESS;
123     }
124
125     int main()
126     {
127         printf("Start program");
128         srand(time(NULL));
129         setbuf(stdout, NULL);
130
131         pthread_t threads[PTHREAD_COUNT];
132         char* names[5] = {"1", "2", "3", "4", "5"};
133
134         int status;
135         int status_addr;
136
137         for (int i = 0; i < PTHREAD_COUNT; i++)
138         {
139             status = pthread_create(&threads[i], NULL, do_pthread, names[i]);
140             if (status != 0) {
141                 printf("main error: can't create thread, status = %d\n",
                        status);
142                 exit(ERROR_CREATE_THREAD);
143             }
144         }
145
146             for (int i = 0; i < PTHREAD_COUNT; i++)
147         {
148             status = pthread_join(threads[i], (void**)&status_addr);
149             if (status != SUCCESS) {
150                 printf("main error: can't join thread, status = %d\n", status);
151                 exit(ERROR_JOIN_THREAD);
152             }
153             printf("joined with address %d\n", status_addr);
154         }
155
156         return 0;
157     }
```

3.5 Вывод

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

4 Экспериментальная часть

В данном разделе ...

4.1 Временные характеристики

4.2 Сравнительный анализ алгоритмов

4.3 Вывод

В данном разделе было ...

Заключение

В рамках выполнения работы решены следующие задачи.

а)

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/`.