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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка загружаемого модуля ядра Linux для мониторинга использования оперативной памяти»

Студент ИУ7-76Б _____ А. А. Петрова
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель _____ Н. Ю. Рязанова
(Подпись, дата) (И.О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

Введение

В настоящее время большую актуальность имеют системы, предоставляющие информацию о ресурсах операционной системы. Имея такие сведения, пользователь может проанализировать состояние системы и нагрузку на неё. Особое внимание уделяется операционным системам с ядром Linux [1]. Ядро Linux возможно изучать благодаря тому, что оно имеет открытый исходный код.

На данный момент существует множество различных утилит и команд для получения информации о свободной и занятой оперативной памяти в Linux. Одни из наиболее известных – это команды free, vmstat, htop, memstat [2]. Также существует приложение GNOME System Monitor, предоставляющее краткую статистику использования системных ресурсов – памяти, процессора, подкачки и сети – в графическом виде [3].

Цель данной работы – реализовать загружаемый модуль ядра Linux, предоставляющий статистику по количеству доступной и занятой оперативной памяти за определенный промежуток времени.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- выполнить постановку задачи;
- проанализировать и сравнить существующие методы и способы её решения;
- описать алгоритм решения поставленной задачи и привести соответствующие схемы и IDEF0-диаграммы;
- разработать ПО в соответствии с заданием;
- проанализировать результаты работы разработанного ПО.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием необходимо разработать загружаемый модуль ядра, предоставляющий статистику по количеству доступной и занятой оперативной памяти за выбранный промежуток времени.

Требования к разрабатываемому ПО:

- ПО должно выдавать количество свободной, доступной и занятой оперативной памяти каждые 10 секунд;
- полученная информация должна записываться в виртуальную файловую систему /proc в файл /proc/monitor/memory.

Ограничения на разрабатываемое ПО:

- ПО не предоставляет информацию о том, чем конкретно занята оперативная память;
- промежуток времени задается внутри программы и может быть изменен только в коде.

1.2 Основные понятия

Свободная память – память, которая в настоящее время ни для чего не используется [4].

Доступная память – память, которая используется, но может быть предоставлена для новых или существующих процессов.

Занятая память – память, которая на данный момент уже используется процессами.

Нижняя область памяти – область, к которой ядро может обращаться напрямую. Все структуры данных ядра находятся в этой области.

Верхняя область памяти – область, доступ к которой осуществляется через косвенные механизмы. Здесь находится кэш данных.

1.3 Анализ методов решения

1.3.1 Файл /proc/meminfo

Одним из способов получения информации об использовании памяти является файл /proc/meminfo [5]. Из /proc/meminfo можно получить информацию о свободной памяти, об используемой (и физической, и swap), а также о разделяемой (shared memory) и буферах.

Основные показатели:

- MemTotal – общий объем оперативной памяти;
- LowFree – объем свободной нижней области памяти;
- HighFree – объем свободной верхней области памяти;
- MemFree – сумма LowFree и HighFree;
- MemAvailable – объем доступной оперативной памяти.

Помимо перечисленных в /proc/meminfo присутствует также множество других показателей, не существенных для данной работы.

Основной недостаток использования такого метода решения поставленной задачи заключается в необходимости постоянного обращения напрямую к указанному файлу и его анализа, что может привести к дополнительным временным затратам.

1.3.2 Структура struct sysinfo

Чтобы избежать чтения и анализа упомянутого выше файла, можно использовать структуру struct sysinfo.

Структура struct sysinfo [6] хранит статистику о всей системе: информацию о времени, прошедшем с начала запуска системы, количество занятой памяти и так далее. В листинге 1 приведено объявление рассматриваемой структуры.

Листинг 1: структура struct sysinfo

```
1  struct sysinfo {
2    __kernel_long_t uptime;   /* Seconds since boot */
3    __kernel_ulong_t loads[3];  /* 1, 5, and 15 minute load averages */
```

```
4     __kernel_ulong_t totalram;  /* Total usable main memory size */
5     __kernel_ulong_t freeram; /* Free memory size */
6     __kernel_ulong_t sharedram; /* Amount of shared memory */
7     __kernel_ulong_t bufferram; /* Memory used by buffers */
8     __kernel_ulong_t totalswap; /* Total swap space size */
9     __kernel_ulong_t freeswap;  /* swap space still available */
10    __u16 procs;        /* Number of current processes */
11    __u16 pad;        /* Explicit padding for m68k */
12    __kernel_ulong_t totalhigh; /* Total high memory size */
13    __kernel_ulong_t freehigh;  /* Available high memory size */
14    __u32 mem_unit;     /* Memory unit size in bytes */
15    char _f[20-2*sizeof(__kernel_ulong_t)-sizeof(__u32)]; /* Padding: libc5
      uses this.. */
16  };
```

Для инициализации этой структуры используется функция si_meminfo(). Стоит отметить, что рассматриваемая структура не содержит информации о доступной памяти в системе. Для того чтобы получить эту информацию, необходимо воспользоваться функцией si_mem_available().

1.4 Загружаемые модули ядра

Одной из особенностей ядра Linux является способность расширения функциональности во время работы, без необходимости компиляции ядра заново. Часть кода, которая может быть добавлена в ядро во время работы, называется **модулем ядра**. Ядро Linux предлагает поддержку большого числа классов модулей. Каждый модуль – это подготовленный объектный код, который может быть загружен в работающее ядро, а позднее может быть выгружен из ядра. Чтобы загрузить модуль в ядро, необходимо воспользоваться командой «insmod name.ko». А для выгрузки модуля – командой «rmmod name.ko».

Каждый модуль ядра сам регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Задача инициализации модуля заключается в подготовке функций модуля для последующего вызова. Функция выхода из модуля вызывается перед выгрузкой модуля из ядра. Функция выхода должна отменить все изменения, сделанные

функцией инициализации, освободить захваченные в процессе работы модуля ресурсы. Для регистрации функций инициализации и выхода используются функции «module_init(func_name)» и «module_exit(func_name)» соответственно.

Модуль связан только с ядром и может вызывать только те функции, которые экспортированы ядром. Для экспорта функций в модуле необходимо использовать «EXPORT_SYMBOL(func_name)».

1.5 Пространство ядра и пространство пользователя

Приложения работают в пользовательском пространстве, а ядро и его модули – в пространстве ядра. Такое разделение пространств – базовая концепция теории операционных систем.

Ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих задач становится возможным только в том случае, если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли и некоторые операции на более низких уровнях не допускаются. Программный код может переключить один уровень на другой только ограниченным числом способов. Все современные процессоры имеют не менее двух уровней защиты.

Ядро Linux выполняется на самом высоком уровне, где разрешено выполнение любых инструкций и доступ к произвольным участкам памяти. А приложения выполняются на самом низком уровне, в котором процессор регулирует прямой доступ к оборудованию и несанкционированный доступ к памяти. Ядро выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием.

1.6 Виртуальная файловая система /proc

Для организации доступа к разнообразным файловым системам в Unix используется промежуточный слой абстракции – **виртуальная файловая система**. С точки зрения программиста, виртуальная файловая система организована как специальный интерфейс. Виртуальная файловая система объявляет API доступа к ней, а реализацию этого API отдает драйверам конкретных файловых систем.

Виртуальная файловая система /proc – специальный интерфейс, с помощью которого можно получить некоторую информацию о ядре в пространство пользователя. /proc отображает в виде дерева каталогов внутренние структуры ядра.

В каталоге /proc в Linux присутствуют несколько деревьев файловой системы. В основном дереве каждый каталог имеет числовое имя, которое соответствует PID процесса. Файлы в этих каталогах соответствуют структуре task_struct, которая имеет вид [7]:

Листинг 2: структура task_struct с наиболее важными полями

```
1  struct task_struct {
2  #ifdef CONFIG_THREAD_INFO_IN_TASK
3    struct thread_info    thread_info;
4  #endif
5
6    unsigned int      __state;
7  ...
8    unsigned int      flags;
9  ...
10 #ifdef CONFIG_SMP
11   int           on_cpu;
12 ...
13   int           recent_used_cpu;
14   int           wake_cpu;
15 #endif
16 ...
17 #ifdef CONFIG_CGROUP_SCHED
```

```
18      struct task_group   *sched_task_group;
19  #endif
20  ...
21      struct sched_info   sched_info;
22      struct list_head    tasks;
23  ...
24  };
```

Так, например, с помощью команды «cat /proc/1/cmdline», можно узнать аргументы запуска процесса с идентификатором, равным единице.

Ядро предоставляет возможность добавить своё дерево в каталог /proc. Внутри ядра объявлена специальная структура struct proc_ops [8]. Эта структура содержит внутри себя указатели на функции чтения файла, записи в файл и прочие, определенные пользователем. В листинге 4 представлено объявление данной структуры в ядре.

Листинг 3: структура struct proc_ops

```
1   struct proc_ops {
2       unsigned int proc_flags;
3       int (*proc_open)(struct inode *, struct file *);
4       ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5       ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
6       ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t
        *);
7       /* mandatory unless nonseekable_open() or equivalent is used */
8       loff_t  (*proc_lseek)(struct file *, loff_t, int);
9       int (*proc_release)(struct inode *, struct file *);
10      __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
11      long  (*proc_ioctl)(struct file *, unsigned int, unsigned long);
12      #ifdef CONFIG_COMPAT
13      long  (*proc_compat_ioctl)(struct file *, unsigned int, unsigned long);
14      #endif
15      int (*proc_mmap)(struct file *, struct vm_area_struct *);
16      unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
        unsigned long, unsigned long, unsigned long);
17  } __randomize_layout;
```

С помощью вызова функций proc_mkdir() и proc_create() в модуле ядра

можно зарегистрировать свои каталоги и файлы в /proc соответственно. Функции copy_to_user() и copy_from_user() реализуют передачу данных из пространства ядра в пространство пользователя и наоборот.

Таким образом, с помощью виртуальной файловой системы /proc можно получать (или передавать) какую-либо информацию из пространства ядра в пространство пользователя (из пространства пользователя в пространство ядра).

Выводы

В этом разделе была проанализирована поставленная задача и методы ее решения. В ходе анализа для получения информации о памяти была выбрана структура struct sysinfo, т. к. такой подход позволяет не работать напрямую с файлом /proc/meminfo. Помимо этого были рассмотрены особенности загружаемых модулей ядра и понятия пространства ядра и пространства пользователя, а также рассмотрен способ взаимодействия этих двух пространств с целью передачи данных из одного в другое.

2 Конструкторская часть

2.1 Архитектура приложения

В состав разрабатываемого программного обеспечения входит один загружаемый модуль ядра, который предоставляет пользователю информацию о загруженности оперативной памяти – ее общее количество, а также количество занятой, свободной и доступной оперативной памяти в данный момент времени.

2.2 Базовые структуры

В листингах 4-5 представлены объявления структур struct proc_ops для работы с виртуальной файловой системой /proc и struct mem_struct для хранения информации о памяти.

Листинг 4: структура struct proc_ops

```
1  static const struct proc_ops mem_ops = {
2    proc_read: seq_read,
3    proc_open: proc_memory_open,
4    proc_release: proc_release,
5  };
```

Листинг 5: структура struct mem_struct

```
1  typedef struct mem_struct {
2    long available;
3    long free;
4    long time_secs;
5  } mem_info_t;
```

Для использования структуры mem_ops необходимо реализовать функции proc_memory_open() и proc_release().

2.3 Разработка алгоритма

На рисунках 1-2 представлены IDEF0-диаграммы алгоритма работы программы.

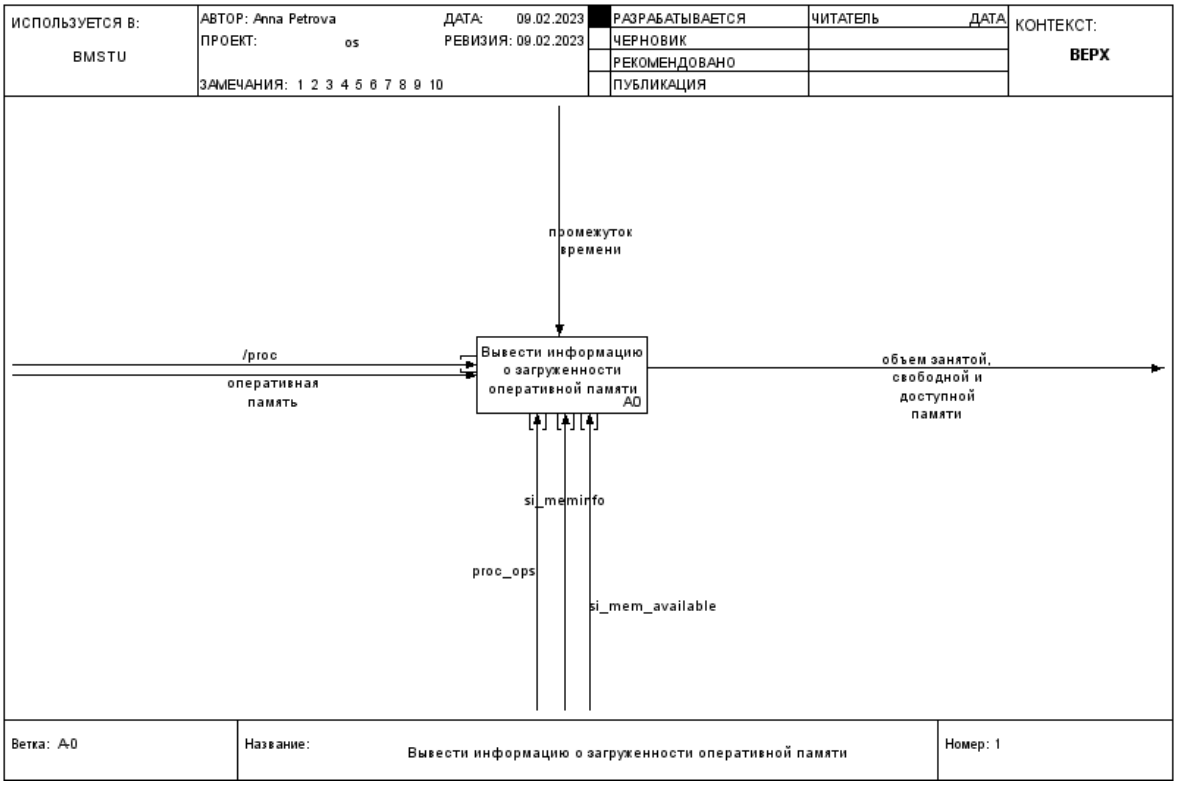

Рисунок 1 – Общая схема работы

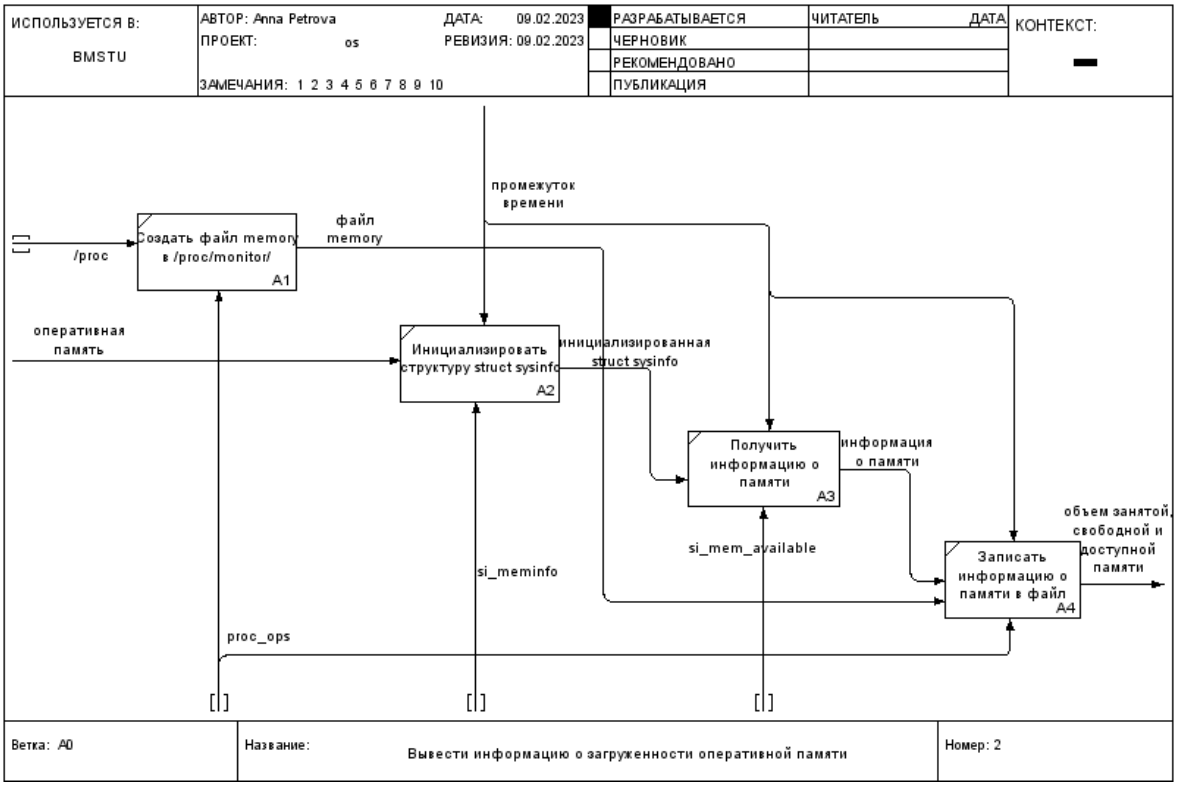

Рисунок 2 – Декомпозиция алгоритма

Выводы

В данном разделе была рассмотрена общая архитектура приложения, базовые структуры, необходимые для разработки, а также были приведены IDEF0-диаграммы алгоритма работы программы.

3 Технологическая часть

3.1 Выбор языка программирования

В качестве языка программирования был выбран язык C [9]. Данный выбор обусловлен тем, что исходный код ядра Linux, все его модули и драйверы написаны на этом языке.

В качестве компилятора был выбран gcc [10].

3.2 Информация о памяти в системе

Для сбора информации о доступной и свободной памяти в системе запускается отдельный поток ядра, который находится в состоянии сна и, просыпаясь каждые 10 секунд, фиксирует эту информацию в результирующий массив. В листинге 6 представлена реализация этого потока, а в листинге 7 его инициализация.

Листинг 6: реализация функции сохраняющей информацию о памяти

```
1  mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
2  int mem_info_calls_cnt;
3
4  int memory_cnt_task_handler_fn(void *args) {
5    struct sysinfo i;
6    struct timespec64 t;
7
8    ENTER_LOG();
9
10   allow_signal(SIGKILL);
11
12   while (!kthread_should_stop()) {
13     si_meminfo(&i);
14
15     ktime_get_real_ts64(&t);
16
17     mem_info_array[mem_info_calls_cnt].free = i.freeram;
18     mem_info_array[mem_info_calls_cnt].available = si_mem_available();
19     mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
20
21     ssleep(10);
```

```
22
23        if (signal_pending(worker_task)) {
24          break;
25        }
26      }
27
28      EXIT_LOG();
29      do_exit(0);
30      return 0;
31    }
```

Листинг 7: инициализация потока ядра

```
1   cpu = get_cpu();
2   worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
      counter thread");
3   kthread_bind(worker_task, cpu);
4
5   if (worker_task == NULL) {
6     cleanup();
7     return -1;
8   }
9
10  wake_up_process(worker_task);
11  return 0;
```

3.3 Детали реализации

В листингах 8-10 представлена реализация точек входа в загружаемый модуль.

Листинг 8: функция инициализации модуля

```
1   static int __init md_init(void) {
2     int rc;
3     int cpu;
4
5     ENTER_LOG();
6
7     if ((rc = proc_init())) {
8       return rc;
9     }
```

```
10
11      cpu = get_cpu();
12      worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
        counter thread");
13      kthread_bind(worker_task, cpu);
14
15      if (worker_task == NULL) {
16        cleanup();
17        return -1;
18      }
19
20      wake_up_process(worker_task);
21
22      printk("%s: module loaded\n", MODULE_NAME);
23      EXIT_LOG();
24
25      return 0;
26    }
```

Листинг 9: функция выхода из модуля

```
1   static void __exit md_exit(void) {
2     cleanup();
3
4     printk("%s: module unloaded\n", MODULE_NAME);
5   }
```

Листинг 10: создание директории и файла в /proc

```
1   static int proc_init(void) {
2     ENTER_LOG();
3
4     if ((proc_root = proc_mkdir(MODULE_NAME, NULL)) == NULL) {
5       goto err;
6     }
7
8     if ((proc_mem_file = proc_create("memory", 066, proc_root, &mem_ops)) ==
      NULL) {
9       goto err;
10    }
11
```

```
12      EXIT_LOG();
13      return 0;
14
15  err:
16      cleanup();
17      EXIT_LOG();
18      return -ENOMEM;
19  }
```

Make файл для компиляции и сборки разработанного загружаемого модуля ядра представлен в листинге 11.

Листинг 11: реализация make файла

```
1   KPATH := /lib/modules/$(shell uname -r)/build
2   MDIR := $(shell pwd)
3
4   obj-m += monitor.o
5   monitor-y := monitor_main.o stat.o log.o
6   EXTRA_CFLAGS=-I$(PWD)/inc
7
8   all:
9       make -C $(KPATH) M=$(MDIR) modules
10
11  clean:
12      make -C $(KPATH) M=$(MDIR) clean
13
14  load:
15      sudo insmod monitor.ko
16
17  unload:
18      sudo rmmod monitor.ko
19
20  info:
21      modinfo monitor.ko
22
23  logs:
24      sudo dmesg | tail -n60 | grep monitor:
```

Выводы

В данном разделе был выбран язык программирования, разработан и протестирован исходный код программы, а также рассмотрены листинги реализованного программного обеспечения.

4 Исследовательская часть

4.1 Результаты работы разработанного ПО

На рисунке 3 представлена информация о разработанном загружаемом модуле ядра и его логи. На рисунке 4 – пример работы модуля, а на рисунке 5 – сравнение результатов его работы с результатами команды «free -m» (у последней все объемы указаны в Мб).

```
a_avortep@avortep:~/prog/os_course/src$ make info
modinfo monitor.ko
filename:       /home/a_avortep/prog/os_course/src/monitor.ko
description:    A utility for monitoring RAM usage
author:         Petrova Anna
license:        GPL
srcversion:     AFA2CB33C0B7E71E1102A2A
depends:
retpoline:      Y
name:           monitor
vermagic:       5.15.0-57-generic SMP mod_unload modversions
a_avortep@avortep:~/prog/os_course/src$ make logs
sudo dmesg | tail -n60 | grep monitor:
[  557.812607] monitor: loading out-of-tree module taints kernel.
[  557.812674] monitor: module verification failed: signature and/or required key missing - tainting kernel
[  557.812874] monitor: function entry md_init | line: 194
[  557.812875] monitor: function entry proc_init | line: 161
[  557.812878] monitor: exit function proc_init | line: 181
[  557.812907] monitor: module loaded
[  557.812908] monitor: exit function md_init | line: 219
[  557.813071] monitor: function entry memory_cnt_task_handler_fn | line: 59
```

Рисунок 3 – Информация о модуле и логи

```
Time 00:22:11
Free:                           3687544 kB
Available:                      5390508 kB
Occupied:       2543792 kB

Time 00:22:22
Free:                           3676760 kB
Available:                      5379732 kB
Occupied:       2554568 kB

Time 00:22:32
Free:                           4506996 kB
Available:                      6207136 kB
Occupied:       1727164 kB

Time 00:22:42
Free:                           4503000 kB
Available:                      6203140 kB
Occupied:       1731160 kB

Time 00:22:52
Free:                           4510588 kB
Available:                      6210736 kB
Occupied:       1723564 kB
a_avortep@avortep:~/prog/os_course/src$
```

Рисунок 4 – Пример работы модуля

```
Time 00:31:35
Free:                           3896304 kB
Available:                      6211480 kB
Occupied:       1722820 kB

Time 00:31:45
Free:                           3886056 kB
Available:                      6201256 kB
Occupied:       1733044 kB

Time 00:31:55
Free:                           4016820 kB
Available:                      6332388 kB
Occupied:       1601912 kB
a_avortep@avortep:~/prog/os_course/src$ free -m
               total        used        free      shared  buff/cache   available
Mem:            7748        1062        3922         220        2763        6184
Swap:           7812           0        7812
a_avortep@avortep:~/prog/os_course/src$
```

Рисунок 5 – Результаты работы команды «free -m»

На рисунке 6 представлена визуализация данных о свободной, доступной и занятой памяти в системе, полученных из разработанного модуля ядра.

Рисунок 6 – Визуализация данных о памяти за 20 минут

4.2 Анализ результатов

На основе рисунка 5 можно сделать вывод, что результаты работы разработанного загружаемого модуля ядра совпадают с результатами работы команды «free -m». Следовательно, результаты корректны.

По рисунку 6 видно, как изменялась загруженность памяти в течение 20 минут. В моменты резкого спада количества занятой оперативной памяти было закрыто несколько приложений, а в моменты роста – наоборот, открыто.

Выводы

В данном разделе были приведены результаты работы разработанного ПО и проведён анализ этих результатов.

В итоге было выяснено, что результаты работы модуля корректны, так как совпадают с результатами работы стандартной команды. Также по приведенному выше графику была выявлена зависимость загруженности оперативной памяти от количества запущенных приложений.

Заключение

Цель курсового проекта достигнута. В ходе проделанной работы был разработан загружаемый модуль ядра, предоставляющий информацию о загруженности оперативной памяти системы.

Для этого были изучены структуры и функции ядра, которые предоставляют информацию о памяти. На основе чего был разработан, а затем реализован алгоритм работы программы.

Помимо этого были также приведены и проанализированы результаты работы ПО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux – Operating System [Электронный ресурс]. – Режим доступа: https://www.linux.org/ (дата обращения: 20.12.2022).

2. Использование оперативной памяти в Linux [Электронный ресурс]. – Режим доступа: https://losst.pro/ispolzovanie-operativnoj-pamyati-linux (дата обращения: 20.12.2022).

3. System Monitor – Apps for GNOME [Электронный ресурс]. – Режим доступа: https://apps.gnome.org/ru/app/gnome-system-monitor/ (дата обращения: 23.12.2022).

4. Free vs. Available Memory in Linux [Электронный ресурс]. – Режим доступа: https://haydenjames.io/free-vs-available-memory-in-linux/ (дата обращения: 23.12.2022).

5. The /proc/meminfo File in Linux [Электронный ресурс]. – Режим доступа: https://www.baeldung.com/linux/proc-meminfo (дата обращения: 24.12.2022).

6. include/uapi/linux/sysinfo.h - Linux source code (v6.1.10) [Электронный ресурс]. – Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/sysinfo.h#L8 (дата обращения: 24.12.2022).

7. /include/linux/sched.h - Linux source code (v6.1.10) [Электронный ресурс]. – Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L737 (дата обращения: 24.12.2022).

8. /include/linux/proc_fs.h - Linux source code (v6.1.10) [Электронный ресурс]. – Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/proc_fs.h#L29 (дата обращения: 24.12.2022).

9. C99 standard note [Электронный ресурс]. – Режим доступа: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf (дата обращения: 17.01.2023).

10. GCC, the GNU Compiler Collection [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/ (дата обращения: 17.01.2023).

ПРИЛОЖЕНИЕ А

Листинг 12: листинг файла monitor_main.c

```
1  #include <linux/module.h>
2  #include <linux/proc_fs.h>
3  #include <linux/time.h>
4  #include <linux/kthread.h>
5
6  #include "memory.h"
7  #include "stat.h"
8
9  MODULE_LICENSE("GPL");
10 MODULE_AUTHOR("Petrova Anna");
11 MODULE_DESCRIPTION("A utility for monitoring RAM usage");
12
13 static struct proc_dir_entry *proc_root = NULL;
14 static struct proc_dir_entry *proc_mem_file = NULL;
15 static struct task_struct *worker_task = NULL;
16
17 static int show_memory(struct seq_file *m, void *v) {
18   print_memory_statistics(m);
19   return 0;
20 }
21
22 static int proc_memory_open(struct inode *sp_inode, struct file *sp_file) {
23   return single_open(sp_file, show_memory, NULL);
24 }
25
26 static int proc_release(struct inode *sp_node, struct file *sp_file) {
27   return 0;
28 }
29
30 mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
31 int mem_info_calls_cnt;
32
33 int memory_cnt_task_handler_fn(void *args) {
34   struct sysinfo i;
35   struct timespec64 t;
36
```

```
37      ENTER_LOG();
38
39      allow_signal(SIGKILL);
40
41      while (!kthread_should_stop()) {
42        si_meminfo(&i);
43
44        ktime_get_real_ts64(&t);
45
46        mem_info_array[mem_info_calls_cnt].free = i.freeram;
47        mem_info_array[mem_info_calls_cnt].available = si_mem_available();
48        mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
49
50        ssleep(10);
51
52        if (signal_pending(worker_task)) {
53          break;
54        }
55      }
56
57      EXIT_LOG();
58      do_exit(0);
59      return 0;
60    }
61
62    #define CHAR_TO_INT(ch) (ch - '0')
63
64    static ktime_t convert_strf_to_seconds(char buf[]) {
65      /* time format: xxhyymzzs. For example: 01h23m45s */
66      ktime_t hours, min, secs;
67
68      hours = CHAR_TO_INT(buf[0]) * 10 + CHAR_TO_INT(buf[1]);
69      min = CHAR_TO_INT(buf[3]) * 10 + CHAR_TO_INT(buf[4]);
70      secs = CHAR_TO_INT(buf[6]) * 10 + CHAR_TO_INT(buf[7]);
71
72      return hours * 60 * 60 + min * 60 + secs;
73    }
74
75    static const struct proc_ops mem_ops = {
76      proc_read: seq_read,
```

```
77      proc_open: proc_memory_open,
78      proc_release: proc_release,
79    };
80
81    static void cleanup(void) {
82      ENTER_LOG();
83
84      if (worker_task) {
85        kthread_stop(worker_task);
86      }
87
88      if (proc_mem_file != NULL) {
89        remove_proc_entry("memory", proc_root);
90      }
91
92      if (proc_root != NULL) {
93        remove_proc_entry(MODULE_NAME, NULL);
94      }
95
96      EXIT_LOG();
97    }
98
99    static int proc_init(void) {
100     ENTER_LOG();
101
102     if ((proc_root = proc_mkdir(MODULE_NAME, NULL)) == NULL) {
103       goto err;
104     }
105
106     if ((proc_mem_file = proc_create("memory", 066, proc_root, &mem_ops)) ==
        NULL) {
107       goto err;
108     }
109
110     EXIT_LOG();
111     return 0;
112
113     err:
114     cleanup();
115     EXIT_LOG();
```

```
116     return -ENOMEM;
117   }
118
119   static int __init md_init(void) {
120     int rc;
121     int cpu;
122
123     ENTER_LOG();
124
125     if ((rc = proc_init())) {
126       return rc;
127     }
128
129     cpu = get_cpu();
130     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
        counter thread");
131     kthread_bind(worker_task, cpu);
132
133     if (worker_task == NULL) {
134       cleanup();
135       return -1;
136     }
137
138     wake_up_process(worker_task);
139
140     printk("%s: module loaded\n", MODULE_NAME);
141     EXIT_LOG();
142
143     return 0;
144   }
145
146   static void __exit md_exit(void) {
147     cleanup();
148
149     printk("%s: module unloaded\n", MODULE_NAME);
150   }
151
152   module_init(md_init);
153   module_exit(md_exit);
```

Листинг 13: листинг файла stat.c

```
1  #include "stat.h"
2
3  #define TASK_STATE_FIELD state
4  #define TASK_STATE_SPEC "%ld"
5
6  static inline long convert_to_kb(const long n) {
7    return n << (PAGE_SHIFT - 10);
8  }
9
10 void print_memory_statistics(struct seq_file *m) {
11   struct sysinfo info;
12   long long secs;
13   long sys_occupied, apps_occupied;
14   int i;
15
16   ENTER_LOG();
17
18   si_meminfo(&info);
19   show_int_message(m, "Memory total: \t%ld kB\n", convert_to_kb(info.
     totalram));
20
21   for (i = 0; i < mem_info_calls_cnt; i++) {
22     secs = mem_info_array[i].time_secs;
23     show_int3_message(m, "\nTime %.2llu:%.2llu:%.2llu\n", (secs / 3600 + 3)
     % 24, secs / 60 % 60, secs % 60);
24     show_int_message(m, "Free:      \t\t\t%ld kB\n", convert_to_kb(
     mem_info_array[i].free));
25     show_int_message(m, "Available: \t\t\t%ld kB\n", convert_to_kb(
     mem_info_array[i].available));
26     sys_occupied = convert_to_kb(info.totalram) - convert_to_kb(
     mem_info_array[i].available);
27     show_int_message(m, "Occupied: \t%ld kB\n", sys_occupied);
28   }
29
30   EXIT_LOG();
31 }
```

Листинг 14: листинг файла log.c

```
1  #include "log.h"
2
3  void show_int_message(struct seq_file *m, const char *const f, const long
     num) {
4    char tmp[256];
5    int len;
6
7    len = snprintf(tmp, 256, f, num);
8    seq_write(m, tmp, len);
9  }
10
11 void show_int3_message(struct seq_file *m, const char *const f, const long
     n1, const long n2, const long n3) {
12   char tmp[256];
13   int len;
14
15   len = snprintf(tmp, 256, f, n1, n2, n3);
16   seq_write(m, tmp, len);
17 }
18
19 void show_str_message(struct seq_file *m, const char *const f, const char *
     const s) {
20   char tmp[256];
21   int len;
22
23   len = snprintf(tmp, 256, f, s);
24   seq_write(m, tmp, len);
25 }
```

Листинг 15: листинг файла memory.h

```
1  #ifndef __MEMORY_H__
2  #define __MEMORY_H__
3
4  #include <linux/kthread.h>
5  #include <linux/delay.h>
6  #include <linux/time.h>
7
8  #include "log.h"
```

```
 9
10  typedef struct mem_struct {
11    long available;
12    long free;
13    long time_secs;
14  } mem_info_t;
15
16  #define MEMORY_ARRAY_SIZE 8640
17  extern mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
18
19  extern int mem_info_calls_cnt;
20
21  #endif
```