



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

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

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

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ НА ТЕМУ:

*«Мониторинг планирования процессов в
UNIX-системах»*

Студент

ИУ7-76Б

П. Ю. Сироткина

(Подпись, дата)

Руководитель

Н. Ю. Рязанова

(Подпись, дата)

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Анализ предметной области	6
1.1 Постановка задачи	6
1.2 Анализ работы планировщика	6
1.2.1 Обычные алгоритмы планирования	7
1.2.2 Алгоритмы реального времени	8
1.3 Анализ структур, предоставляющих информацию о процессах .	9
1.3.1 struct task_struct	9
1.3.2 struct sched_info	12
1.4 Передача информации из пространства ядра в пространство пользователя	13
1.4.1 Файловая система proc	13
2 Конструкторский раздел	15
2.1 Последовательность выполняемых в ПО преобразований	15
2.2 Алгоритм определения информации о планировании процессов в пространстве ядра	17
2.3 Алгоритм получения информации о планировании процессов в пространстве пользователя	18
2.4 Структура разрабатываемого ПО	19
3 Технологический раздел	20
3.1 Выбор языка и среды программирования	20
3.2 Описание объявленных структур и кодов функций	20
3.3 Makefile	23
4 Исследовательский раздел	24
4.1 Технические характеристики	24
4.2 Демонстрация работы ПО	24
4.3 Исследование планирования процесса стандартной игры	26
4.4 Исследование планирования процесса воспроизведения видео- файла	27

4.5	Исследование планирования процесса migration	28
ЗАКЛЮЧЕНИЕ		30
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ		31
ПРИЛОЖЕНИЕ А Исходный код		32

ВВЕДЕНИЕ

Принятие решения о том, как должны использоваться ресурсы различными процессами, возложено на операционную систему. [1] Планировщик является компонентом ОС, определяющим, какой из процессов должен выполняться в данный момент времени и как долго он может занимать процессор.

Задача мониторинга состояния системы, в том числе планирования процессов на выполнение, имеет большую актуальность. Получив и проанализировав полученную таким образом информацию, а также предприняв соответствующие действия, можно добиться повышения интерактивности системы.

Т.к. необходимая информация содержится в структурах ядра (`struct task_struct`, `struct sched_info`), программное обеспечение будет разработано в виде загружаемого модуля ядра. Для передачи информации из пространства ядра в пространство пользователя и наоборот наиболее часто используется виртуальная файловая система `proc`.

Данная курсовая работа посвящена определению информации о планировании процессов в UNIX-системах.

1 Анализ предметной области

1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу по дисциплине «Операционные системы» требуется разработать загружаемый модуль ядра, предоставляющий пользователю возможность получения информации о приоритетах процессов в системе, количестве запусков на выполнение, количестве времени ожидания процессов в очереди на выполнение, времени последнего добавления процесса в очередь.

Для решения поставленной задачи необходимо:

- проанализировать работу планировщика в linux;
- провести анализ структур, предоставляющих необходимую информацию о процессах;
- провести анализ и выбрать способ передачи информации из пространства ядра в пространство пользователя;
- разработать алгоритмы и структуру программного обеспечения;
- исследовать разработанное программное обеспечение.

1.2 Анализ работы планировщика

Планировщик является компонентом ОС, определяющим, какой из процессов должен выполняться в данный момент времени и как долго он может занимать процессор. Каждому процессу назначается алгоритм планирования и статический приоритет планирования `sched_priority` (значение от 0 до 99). Планировщик принимает решение на основе данных об алгоритме планирования и статическом приоритете всех процессов.

Планировщик хранит в памяти списки всех работающих процессов для каждого возможного значения `sched_priority`. Чтобы определить, какой процесс выполнить следующим, планировщик ищет непустой список с самым высоким статическим приоритетом и выбирает процесс из начала списка.

В Linux алгоритмы планирования делятся на обычные алгоритмы и алгоритмы реального времени. Алгоритм планирования определяет, в какое

место списка будет добавлен процесс с тем же статическим процессом, что и другие, и как он будет перемещаться внутри этого списка. [2]

1.2.1 Обычные алгоритмы планирования

Для процессов, которые планируются одним из обычных алгоритмов планирования, значение `sched_priority` не используется. Ранее использовался планировщик $O(1)$, основывающийся на очередях выполнения, а начиная с версии с 2.6.23, распределение ресурсов осуществляется при помощи планировщика CFS (Completely Fair Scheduler, «абсолютно справедливый планировщик»): для хранения процессов использует красно-черное дерево, реализующее «временную шкалу» (поле `p->se.vruntime`) будущего выполнения задачи.

Когда планировщик запускается для запуска нового процесса:

- Выбирается крайний левый узел дерева планирования (так как у него будет наименьшее затраченное время выполнения) и отправляется на выполнение.
- Если процесс завершает выполнение, он удаляется из системы и дерева планирования.
- Если процесс достигает максимального времени выполнения или останавливается иным образом (добровольно или посредством прерывания), он повторно вставляется в дерево планирования на основе нового затраченного времени выполнения.
- Новый крайний левый узел будет затем выбран из дерева, повторяя итерацию.

Существующие обычные алгоритмы планирования:

1. Алгоритм `SCHED_OTHER` - стандартный планировщик Linux с разделением времени, используемый для обычных задач. Задача для запуска выбирается из списка со статическим приоритетом на основе динамического приоритета, который определяется только внутри этого списка, основываясь на уровне `nice` (устанавливаемом системным вызовом `nice` или `setpriority`).

2. Алгоритм `SCHED_BATCH` - планировщик для пакетных процессов. Задачи, запланированные с помощью `SCHED_BATCH`, считаются неинтерактивными, но привязанными к ЦП и оптимизированными для пропускной способности. Таким образом, этот алгоритм более удобен для кэширования.
3. Алгоритм `SCHED_IDLE` - планировщик задач с низким приоритетом.

1.2.2 Алгоритмы реального времени

Для процессов, которые планируются одним из данных алгоритмов планирования, значение `sched_priority` лежит в диапазоне от 1 (min) до 99 (max).

1. Алгоритм `SCHED_FIFO` - планировщик без квантования времени. Планировщик проверяет список задач `SCHED_FIFO` и запускает задачу с наивысшим приоритетом. Задача будет выполняться до тех пор, пока не завершит работу или не будет вытеснена другой задачей с более высоким приоритетом.
2. Алгоритм `SCHED_RR` - циклический вариант (Round-Robin) алгоритма `SCHED_FIFO`. Задачи с одинаковым приоритетом будут последовательно выполняться в рамках выделенного интервала. По истечении выделенного интервала задача перемещается в конец списка с тем же приоритетом.
3. Алгоритм `SCHED_DEADLINE` - реализует алгоритм планирования по ближайшему сроку завершения (EDF), основанный на идее выбора для выполнения из очереди ожидающих процессов задачи, наиболее близкой к истечению крайнего расчётного времени.

Для установки и получения политики планирования процесса используются функции `sched_setscheduler` и `sched_getscheduler` соответственно, которые определены в файле `<include/linux/sched.h>`.

1.3 Анализ структур, предоставляющих информацию о процессах

1.3.1 struct task_struct

Для хранения информации о процессах в ядре используется циклический двухсвязный список записей struct task_struct. Эта структура определена в файле <include/linux/sched.h>. В листинге 1.1 приведены некоторые поля данной структуры.

Листинг 1.1 – struct task_struct

```
1 struct task_struct
2 {
3     ...
4     unsigned int __state ;
5     int prio;
6     unsigned int rt_priority;
7     int static_prio;
8     int normal_prio;
9     int policy;
10    struct sched_info sched_info;
11    pid_t pid;
12    char comm[TASK_COMM_LEN];
13    ...
14 }
```

Предоставляемая информация:

- поле __state - состояние процесса.
- поле pid - уникальный идентификатор процесса;
- поле comm - имя исполняемого файла;
- поле policy - политика планирования;
- поле rt_priority - приоритет планирования реального времени, число в диапазоне от 1 до 99 для процессов реального времени, 0 для остальных.
- поле normal_prio - нормальный приоритет процесса, зависит от статического приоритета и политики планировщика задач. Для процессов не реального времени данное значение равняется значению статического

приоритета `static_prio`. Для процессов реального времени данное значение равняется значению, вычисленному с использованием максимального значения приоритета процесса реального времени и `rt_priority`;

- поле `static_priority` - статический приоритет процесса, не изменяется ядром при работе планировщика, однако оно может быть изменено с использованием макроса `NICE_TO_PRIO` (`<include/linux/sched/prio.h>`);
- поле `prio` - значение, которое использует планировщик задач при выборе процесса. Может принимать значения от 0 до 139. Чем ниже значение данной переменной, тем выше приоритет процесса;
- поле `sched_info` - экземпляр соответствующей структуры, определение которой будет рассмотрено далее;

Для мониторинга будут использованы следующие поля: `pid`, `comm`, `policy`, `prio`, `rt_priority`, `normal_priority`, `static_priority`, `sched_info`.

В листинге 1.2 приведен список макросов, определяющих возможные состояния процесса.

Листинг 1.2 – Список макросов, определяющих возможное состояние процесса

```
1 #define TASK_RUNNING 0x00000000
2 #define TASK_INTERRUPTIBLE 0x00000001
3 #define TASK_UNINTERRUPTIBLE 0x00000002
4 #define __TASK_STOPPED 0x00000004
5 #define __TASK_TRACED 0x00000008
6 #define EXIT_DEAD 0x00000010
7 #define EXIT_ZOMBIE 0x00000020
8 #define EXIT_TRACE ( EXIT_ZOMBIE | EXIT_DEAD )
9 #define TASK_PARKED 0x00000040
10 #define TASK_DEAD 0x00000080
11 #define TASK_WAKEKILL 0x00000100
12 #define TASK_WAKING 0x00000200
13 #define TASK_NOLOAD 0x00000400
14 #define TASK_NEW 0x00000800
15 #define TASK_RTLOCK_WAIT 0x00001000
16 #define TASK_FREEZABLE 0x00002000
17 #define __TASK_FREEZABLE_UNSAFE (0x00004000 * IS_ENABLED(CONFIG_LOCKDEP))
18 #define TASK_FROZEN 0x00008000
19 #define TASK_STATE_MAX 0x00010000
```

В листинге 1.3 представлена функция вычисления нормального приоритета.

Листинг 1.3 – Функция вычисления нормального приоритета

```
1 static inline int __normal_prio(struct task_struct *p)
2 {
3     return p->static_prio;
4 }
5
6 /*
7  * Calculate the expected normal priority: i.e. priority
8  * without taking RT-inheritance into account. Might be
9  * boosted by interactivity modifiers. Changes upon fork,
10 * setprio syscalls, and whenever the interactivity
11 * estimator recalculates.
12 */
13 static inline int normal_prio(struct task_struct *p)
14 {
15     int prio;
16
17     if (task_has_rt_policy(p))
18         prio = MAX_RT_PRIO-1 - p->rt_priority;
19     else
20         prio = __normal_prio(p);
21     return prio;
22 }
```

В листинге 1.4 представлена функция, которая определяет, является ли процесс задачей реального времени, с использованием значения приоритета.

Листинг 1.4 – Функция определения процесса реального времени

```
1 #define MAX_USER_RT_PRIO    100
2 #define MAX_RT_PRIO        MAX_USER_RT_PRIO
3
4 static inline int rt_task(struct task_struct *p)
5 {
6     return rt_prio(p->prio);
7 }
8
9 static inline int rt_prio(int prio)
10 {
11     if (unlikely(prio < MAX_RT_PRIO))
12         return 1;
13     return 0;
14 }
```

В листинге 1.5 представлена функция, которая определяет, является ли процесс задачей реального времени, на основе значения поля `policy`.

Листинг 1.5 – Функция определения процесса реального времени

```
1 static inline bool task_is_realtime(struct task_struct *tsk)
2 {
3     int policy = tsk->policy;
4
5     if (policy == SCHED_FIFO || policy == SCHED_RR)
6         return true;
7     if (policy == SCHED_DEADLINE)
8         return true;
9
10    return false;
11 }
```

Для итерирования по списку процессов внутри ядра могут использоваться макросы `next_task` или `for_each_process`, определенные в файле `<include/linux/sched/signal.h>`.

1.3.2 struct sched_info

Структура `sched_info` предоставляет информацию о планировании процессов. Данная структура определена в файле `<include/linux/sched.h>` и представлена в листинге 1.6.

Листинг 1.6 – struct sched_info

```
1 struct sched_info
2 {
3     /* Cumulative counters: */
4
5     /* # of times we have run on this CPU: */
6     unsigned long          pcount;
7
8     /* Time spent waiting on a runqueue: */
9     unsigned long long     run_delay;
10
11    /* Timestamps: */
12
13    /* When did we last run on a CPU? */
14    unsigned long long      last_arrival;
15
16    /* When were we last queued to run? */
17    unsigned long long      last_queued;
18 };
```

Предоставляемая информация:

- поле `rcount` содержит количество запусков процесса на исполнение центральным процессором;
- поле `run_delay` отражает время (в тиках. после начальной загрузки системы), которое процесс проводит в состоянии ожидания на выполнение;
- поле `last_arrival` отражает время, когда процесс был запущен центральным процессором на выполнение в последний раз;
- поле `last_queued` отражает время, когда процесс был добавлен в очередь на исполнение в последний раз.

Все эти поля будут использованы для мониторинга состояния системы.

1.4 Передача информации из пространства ядра в пространство пользователя

1.4.1 Файловая система `proc`

Файловая система `proc` создана для того, чтобы в режиме пользователя получать информацию о процессах и ресурсах, которые используют эти процессы. ФС `proc` формирует интерфейс, позволяющий обращаться к структурам ядра.

В `proc` можно создавать свои файлы, ссылки, директории. Для этого в ядре предоставляется структура `proc_dir_entry`, определенная в файле `<fs/proc/internal.h>`. Для определения обратных вызовов чтения и записи предоставляется структура `proc_ops`. Данная структура определена в файле `<include/linux/proc_fs.h>`, наиболее важные поля которой представлены в листинге 1.7.

Листинг 1.7 – Структура `proc_ops`

```
1 struct proc_ops {
2     ...
3
4     int (*proc_open)(struct inode *, struct file *);
5
6     ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
7 }
```

```

8      ssize_t (*proc_write)(struct file *, const char __user *, size_t,
9                             loff_t *);
10     int (*proc_release)(struct inode *, struct file *);
11
12     ...
13 };

```

Использование функций `proc_create` и `remove_proc_entry`, определенных в файле `<include/linux/proc_fs.h>`, позволяет регистрировать и отменять регистрацию файла в `proc` соответственно.

Для взаимодействия ядра с приложениями используется функция `copy_to_user`, определение которой представлено в файле `<include/linux/uaccess.h>`. Данная функция копирует блоки данных из ядра в пространство пользователя. Функция возвращает количество байт, которые не могут быть скопированы.

Выводы

В результате анализа были определены структуры, содержащие необходимую информацию о процессах (`struct task_struct`, `struct sched_info`), а также был определен способ передачи данных из пространства ядра в пространство пользователя (использование VFS `proc`).

2 Конструкторский раздел

2.1 Последовательность выполняемых в ПО преобразований

На рисунках 2.1 и 2.2 представлены диаграммы IDEF0 уровня 0 и 1 соответственно.

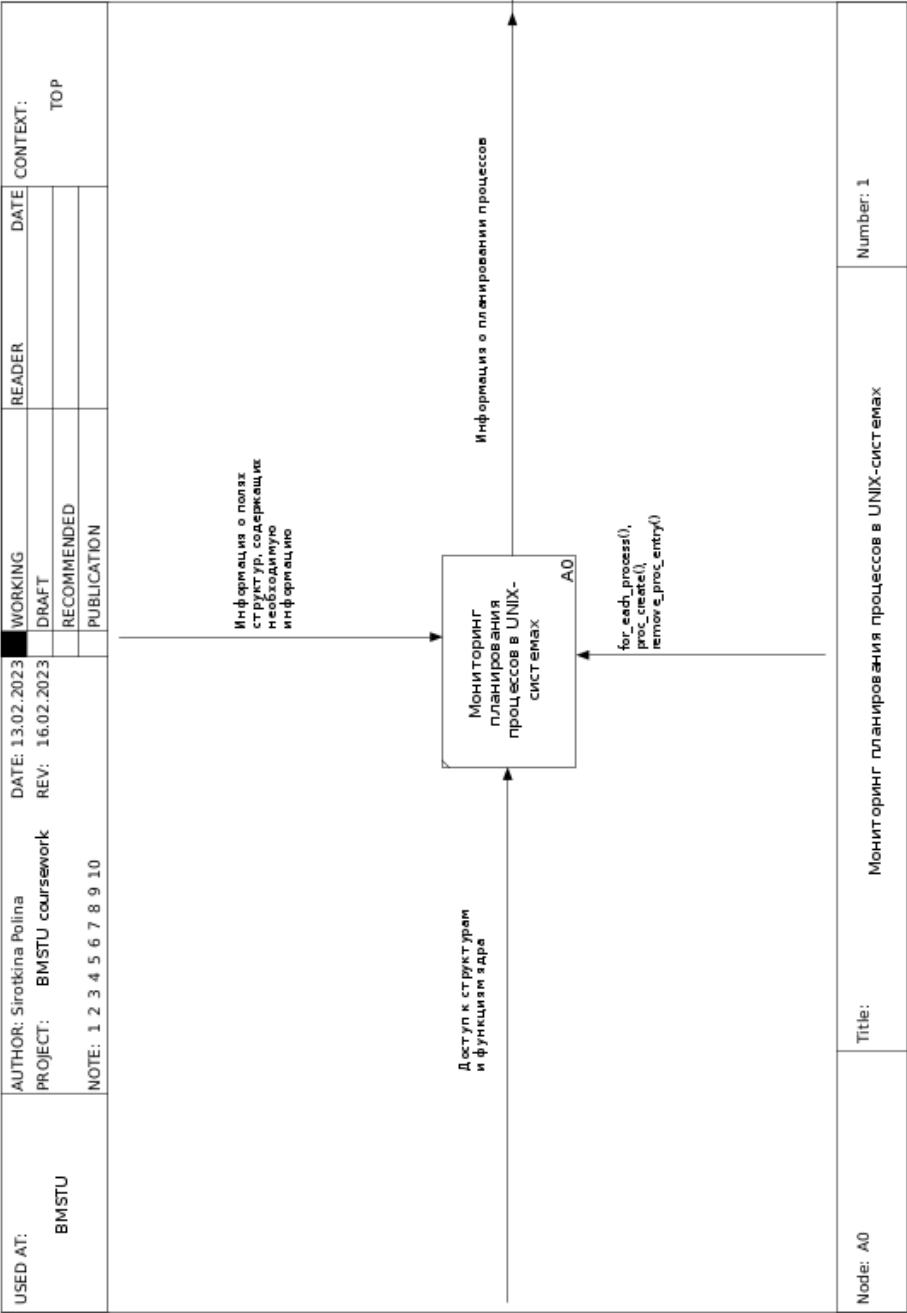


Рисунок 2.1 – Диаграмма IDEF0 уровня 0

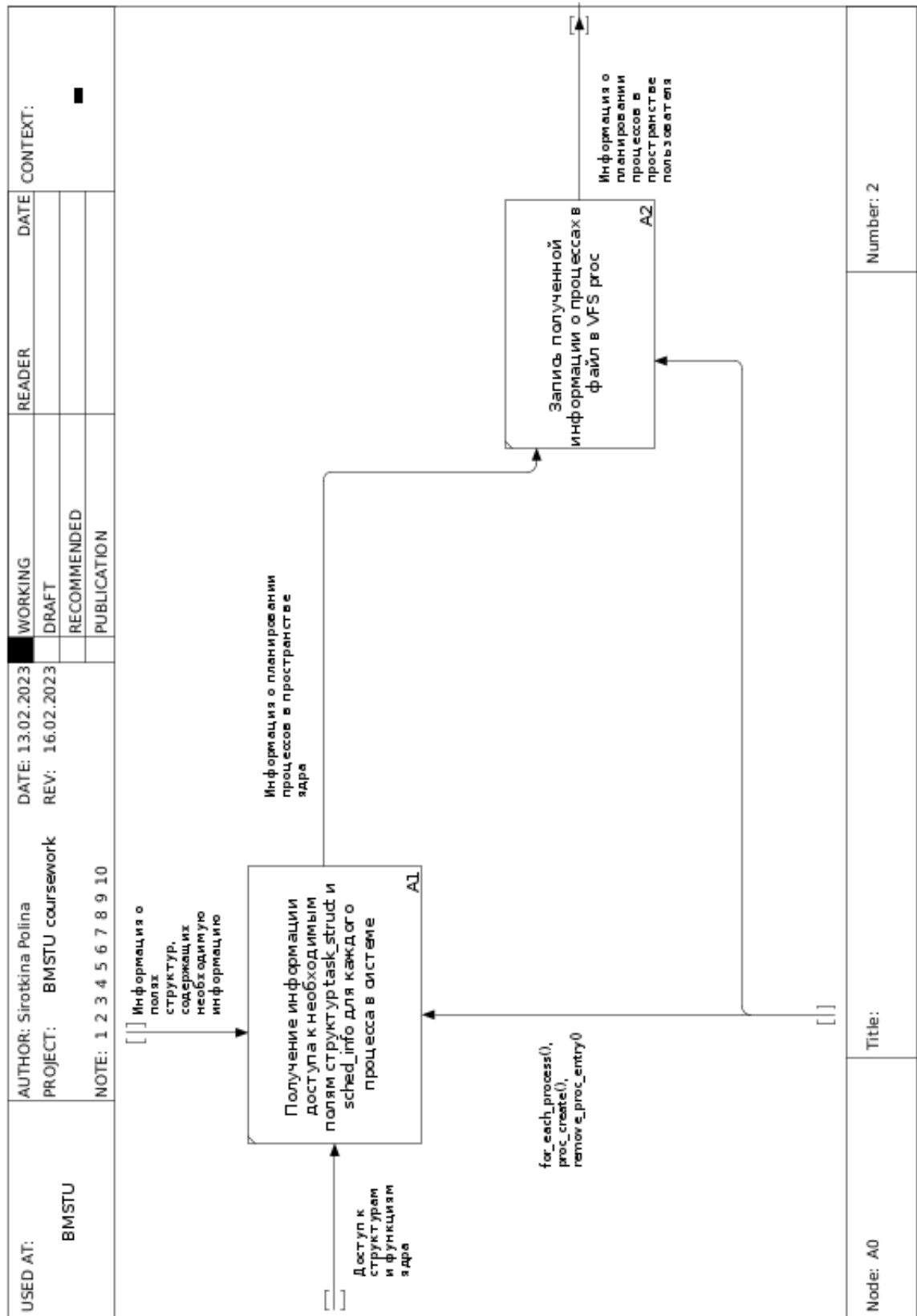


Рисунок 2.2 – Диаграмма IDEF0 уровня 1

2.2 Алгоритм определения информации о планировании процессов в пространстве ядра

На рисунке 2.3 представлен алгоритм определения информации о планировании процессов в пространстве ядра.

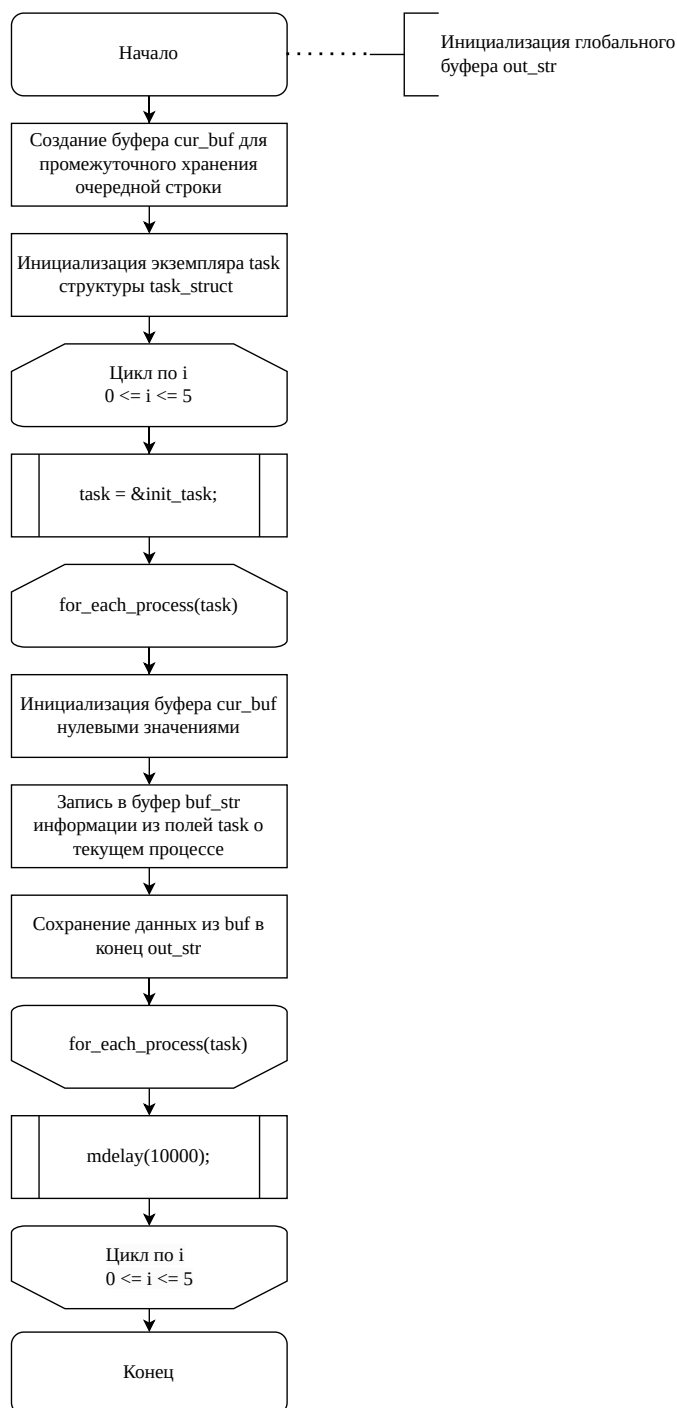


Рисунок 2.3 – Алгоритм определения информации о планировании процессов

2.3 Алгоритм получения информации о планировании процессов в пространстве пользователя

На рисунке 2.4 представлен алгоритм получения информации о планировании процессов.

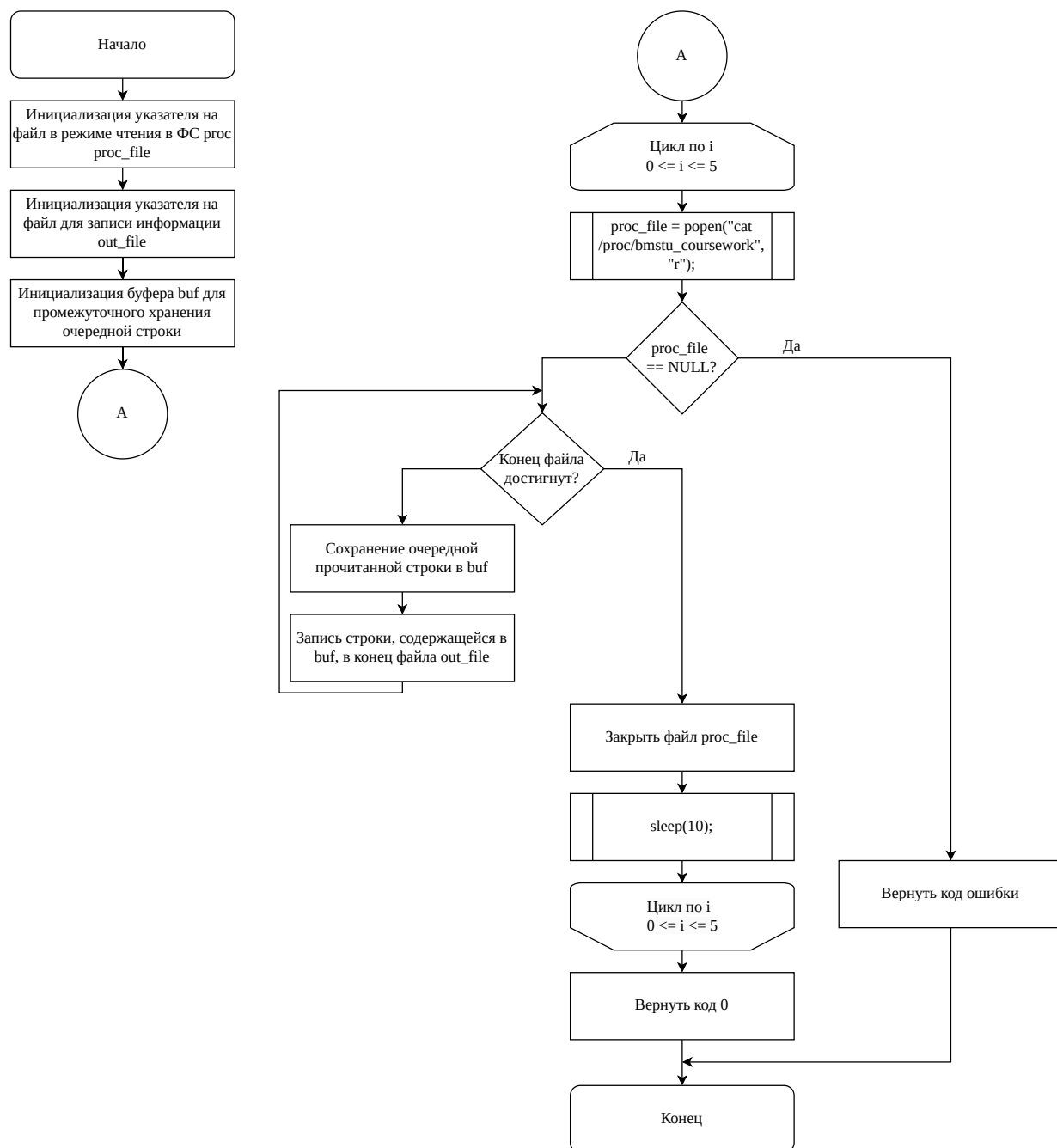


Рисунок 2.4 – Алгоритм получения информации о планировании процессов в пространстве пользователя

2.4 Структура разрабатываемого ПО

На рисунке 2.5 представлена структура разрабатываемого программного обеспечения.

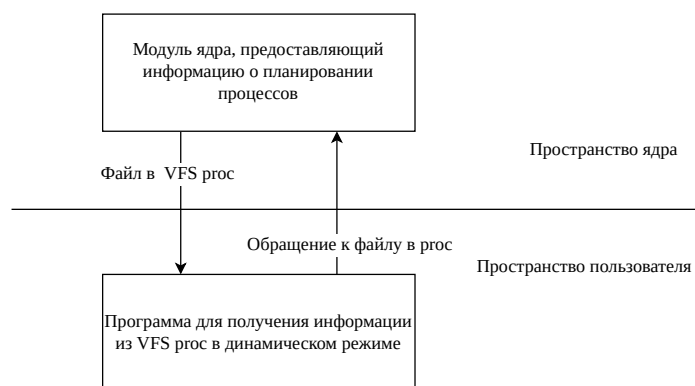


Рисунок 2.5 – Структура разрабатываемого ПО

3 Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования для написания загружаемого модуля ядра был выбран язык C, т.к. исходный код ядра рассматриваемой операционной системы также написан на этом языке. В качестве компилятора был выбран gcc. [3; 4]

В качестве среды программирования была выбрана среда Visual Studio Code, т.к. она бесплатна в использовании, кроссплатформенная и имеет множество расширений. [5]

3.2 Описание объявленных структур и кодов функций

В листинге 3.1 представлена реализация функции определения информации о процессах из структур ядра.

Листинг 3.1 – Функция определения информации о процессах

```
1 static int print_info(void *arg)
2 {
3     struct task_struct *task;
4     int i;
5     char cur_buf[TEMP_BUF_SIZE];
6     for (i = 0; i < REPEAT; i++)
7     {
8         task = &init_task;
9
10        memset(cur_buf, 0, TEMP_BUF_SIZE);
11        snprintf(cur_buf, TEMP_BUF_SIZE,
12        "===== ITERATION #%d =====\n",
13        i+1);
14
15        check_buf(cur_buf, out_str, BUF_SIZE);
16        strcat(out_str, cur_buf);
17
18        for_each_process(task) {
19            memset(cur_buf, 0, TEMP_BUF_SIZE);
20            snprintf(cur_buf, TEMP_BUF_SIZE,
21            "PID: %d;\n"
22            "Process comm: %s;\n"
23            "Policy: %d;\n"
24            "Prio: %d\n"
25            "Static_prio: %d;\n"
```

```

26         "Normal_prio: %d;\n"
27         "Rt_prio: %d;\n"
28         "Delay: %lld;\n"
29         "Calls: %ld;\n"
30         "Time of last run: %lld;\n"
31         "Last queued: %lld;\n\n\n",
32         task->pid, task->comm,
33         task->policy,
34         task->prio, task->static_prio,
35         task->normal_prio,
36         task->rt_priority,
37         task->sched_info.run_delay,
38         task->sched_info.pcount,
39         task->sched_info.last_arrival,
40         task->sched_info.last_queued);
41
42         check_buf(cur_buf, out_str, BUF_SIZE);
43         strcat(out_str, cur_buf);
44     }
45     mdelay(DELAY_MS);
46 }
47 return 0;
48 }

```

В листинге 3.2 представлено определение структуры `proc_ops`.

Листинг 3.2 – Определение функций для работы с файлом в `proc`

```

1  static int my_open(struct inode *inode, struct file *file)
2  {
3      printk(KERN_INFO "%s my_open called.\n", DELIMITER);
4
5      if (!try_module_get(THIS_MODULE))
6      {
7          printk(KERN_INFO "%s error while try_module_get().\n", DELIMITER);
8          return -EFAULT;
9      }
10
11     return 0;
12 }
13
14 static ssize_t my_read(struct file *file, char __user *buf, size_t count,
15                        loff_t *offp)
16 {
17     ssize_t out_str_len = strlen(out_str);
18
19     printk(KERN_INFO "%s my_read called.\n", DELIMITER);
20
21     if (copy_to_user(buf, out_str, strlen(out_str)))

```

```

22     {
23         printk(KERN_ERR "%s error while copy_to_user().\n", DELIMITER);
24         return -EFAULT;
25     }
26
27     memset(out_str, 0, BUF_SIZE);
28
29     return out_str_len;
30 }
31
32 static ssize_t my_write(struct file *file, const char __user *buf,
33                        size_t len, loff_t *offp)
34 {
35     printk(KERN_INFO "%s my_write called.\n", DELIMITER);
36     return 0;
37 }
38
39 static int my_release(struct inode *inode, struct file *file)
40 {
41     printk(KERN_INFO "%s my_release called.\n", DELIMITER);
42     module_put(THIS_MODULE);
43     return 0;
44 }
45
46 static struct proc_ops my_ops = {
47     proc_read: my_read,
48     proc_write: my_write,
49     proc_open: my_open,
50     proc_release: my_release
51 };

```

В листинге 3.3 представлена реализация дополнительной функции получения в динамическом виде информации из файла в ФС `proc` в режиме пользователя.

Листинг 3.3 – Функция получения информации в режиме пользователя

```

1  int main(int argc, char *argv[])
2  {
3      FILE *proc_file = NULL;
4      FILE *out_file = fopen("out.txt", "w");
5
6      char buf[BUF_SIZE] = { '\0' };
7
8      for (int i = 0; i < REPEAT; i++)
9      {
10         proc_file = popen("cat /proc/bmstu_coursework", "r");
11

```

```

12         if (proc_file == NULL)
13         {
14             printf("Error while popen.");
15             return EXIT_FAILURE;
16         }
17
18         fgets(buf, sizeof(buf), proc_file);
19         fprintf(out_file, "%s", buf);
20
21         while (fgets(buf, sizeof(buf), proc_file) != NULL)
22         {
23             fprintf(out_file, "%s", buf);
24         }
25
26         pclose(proc_file);
27         sleep(DELAY);
28     }
29
30     fclose(out_file);
31
32     return EXIT_SUCCESS;
33 }

```

Полный код загружаемого модуля ядра и дополнительных функций будет приведен в приложении.

3.3 Makefile

В листинге 3.4 представлено содержание make файла для сборки проекта.

Листинг 3.4 – Makefile

```

1  CURRENT = $(shell uname -r)
2  KDIR = /lib/modules/$(CURRENT)/build
3  PWD = $(shell pwd)
4  TARGET = md
5  obj-m := $(TARGET).o
6  default:
7      $(MAKE) -C $(KDIR) M=$(PWD) modules
8  getter:
9      gcc getter.c -Wall -Werror -o getter.exe
10 clean:
11     @rm -f *.o *.cmd *.flags *.mod.c *.order
12     @rm -f *.*.cmd *~ *.*~ TODO.*
13     @rm -fR .tmp*
14     @rm -rf .tmp_versions
15 disclean: clean
16     @rm -f *.ko *.symvers *.mod *.exe

```

4 Исследовательский раздел

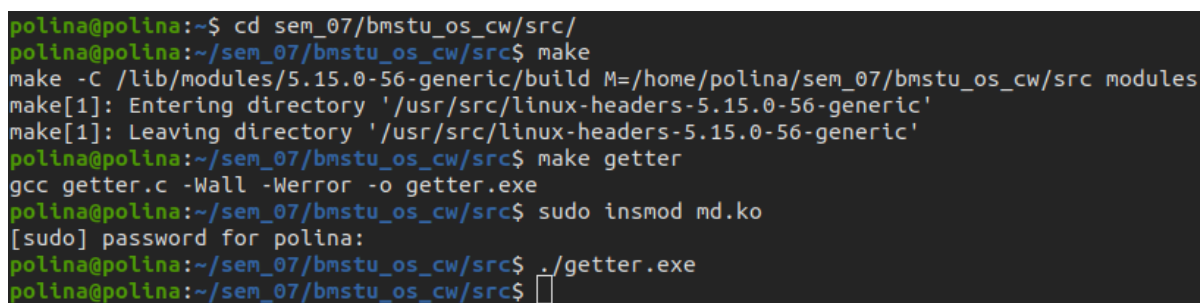
4.1 Технические характеристики

Технические характеристики машины, на которой выполнялось исследование:

- операционная система: Ubuntu Linux 20.04 x86-64;
- оперативная память: 16 Гб;
- процессор: AMD(R) Ryzen(TM) 5 4500U CPU @ 2.3 CHz.

4.2 Демонстрация работы ПО

На рисунке 4.1 представлена последовательность действий для сборки и запуска проекта. Результат работы сохраняется в текстовый файл.



```
polina@polina:~$ cd sem_07/bmstu_os_cw/src/
polina@polina:~/sem_07/bmstu_os_cw/src$ make
make -C /lib/modules/5.15.0-56-generic/build M=/home/polina/sem_07/bmstu_os_cw/src modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-56-generic'
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-56-generic'
polina@polina:~/sem_07/bmstu_os_cw/src$ make getter
gcc getter.c -Wall -Werror -o getter.exe
polina@polina:~/sem_07/bmstu_os_cw/src$ sudo insmod md.ko
[sudo] password for polina:
polina@polina:~/sem_07/bmstu_os_cw/src$ ./getter.exe
polina@polina:~/sem_07/bmstu_os_cw/src$ █
```

Рисунок 4.1 – Сборка и запуск проекта

В листинге 4.1 представлен фрагмент вывода информации о планировании процессов из созданного текстового файла.

Листинг 4.1 – Пример работы программы

```
1 PID: 46;
2 Process comm: ksoftirqd/5;
3 Policy: 0;
4 Prio: 120
5 Static_prio: 120;
6 Normal_prio: 120;
7 Rt_prio: 0;
8 Delay: 214349395;
9 Calls: 4978;
10 Time of last run: 861522808657;
11 Last queued: 0;
12
13
14 PID: 48;
15 Process comm: kworker/5:0H;
16 Policy: 0;
17 Prio: 100
18 Static_prio: 100;
19 Normal_prio: 100;
20 Rt_prio: 0;
21 Delay: 1621341;
22 Calls: 5;
23 Time of last run: 1297277643;
24 Last queued: 0;
25
26
27 PID: 49;
28 Process comm: kdevtmpfs;
29 Policy: 0;
30 Prio: 120
31 Static_prio: 120;
32 Normal_prio: 120;
33 Rt_prio: 0;
34 Delay: 4399566;
35 Calls: 274;
36 Time of last run: 584169101043;
37 Last queued: 0;
```


4.3 Исследование планирования процесса стандартной игры

В листинге 4.2 представлен фрагмент вывода информации о планировании процессов во время выполнения стандартной игры. Здесь и далее рассматривается только 1 итерация вывода, т.к. на протяжении всех итераций приоритет остается одинаковым.

Листинг 4.2 – Пример работы программы при воспроизведении игры

```
1 PID: 3331;
2 Process comm: gnome-mahjongg;
3 Policy: 0;
4 Prio: 120
5 Static_prio: 120;
6 Normal_prio: 120;
7 Rt_prio: 0;
8 Delay: 15619408;
9 Calls: 704;
10 Time of last run: 48891789502;
11 Last queued: 0;
12
13 PID: 3331;
14 Process comm: gnome-mahjongg;
15 Policy: 0;
16 Prio: 120
17 Static_prio: 120;
18 Normal_prio: 120;
19 Rt_prio: 0;
20 Delay: 15683321;
21 Calls: 706;
22 Time of last run: 56790291756;
23 Last queued: 0;
24
25 PID: 3331;
26 Process comm: gnome-mahjongg;
27 Policy: 0;
28 Prio: 120
29 Static_prio: 120;
30 Normal_prio: 120;
31 Rt_prio: 0;
32 Delay: 15927676;
33 Calls: 708;
34 Time of last run: 66839077645;
35 Last queued: 0;
```

Приоритет соответствующего процесса равен 120, а поле policy = 0, что

означает, что данный процесс был запланирован с использованием политики SCHED_OTHER, что означает, что процесс игры не является задачей реального времени.

4.4 Исследование планирования процесса воспроизведения видеофайла

В листинге 4.3 представлен фрагмент вывода информации о планировании процессов во время воспроизведения видеофайла.

Листинг 4.3 – Пример работы программы при воспроизведении видеофайла

```
1 PID: 6094;
2 Process comm: vlc;
3 Policy: 0;
4 Prio: 120
5 Static_prio: 120;
6 Normal_prio: 120;
7 Rt_prio: 0;
8 Delay: 158806408;
9 Calls: 1141;
10 Time of last run: 149508520726;
11 Last queued: 0;
12
13 PID: 6094;
14 Process comm: vlc;
15 Policy: 0;
16 Prio: 120
17 Static_prio: 120;
18 Normal_prio: 120;
19 Rt_prio: 0;
20 Delay: 158806408;
21 Calls: 1141;
22 Time of last run: 149508520726;
23 Last queued: 0;
24
25 PID: 6094;
26 Process comm: vlc;
27 Policy: 0;
28 Prio: 120
29 Static_prio: 120;
30 Normal_prio: 120;
31 Rt_prio: 0;
32 Delay: 158806408;
33 Calls: 1141;
34 Time of last run: 149508520726;
35 Last queued: 0;
```

Приоритет соответствующего процесса равен 120, а поле `policy = 0`, что означает, что данный процесс был запланирован с использованием политики `SCHED_OTHER`, что означает, что процесс воспроизведения видеофайла не является задачей реального времени

4.5 Исследование планирования процесса migration

Процесс migration ядра распределяет рабочую нагрузку между ядрами центрального процессора. В листинге 4.4 представлен фрагмент вывода информации о планировании данного процесса (несколько итераций).

Листинг 4.4 – Пример работы программы

```
1  PID: 15;
2  Process comm: migration/0;
3  Policy: 1;
4  Prio: 0
5  Static_prio: 120;
6  Normal_prio: 0;
7  Rt_prio: 99;
8  Delay: 37010872;
9  Calls: 283;
10 Time of last run: 476177189955;
11 Last queued: 0;
12
13 PID: 15;
14 Process comm: migration/0;
15 Policy: 1;
16 Prio: 0
17 Static_prio: 120;
18 Normal_prio: 0;
19 Rt_prio: 99;
20 Delay: 37052270;
21 Calls: 289;
22 Time of last run: 489549804859;
23 Last queued: 0;
24
25 PID: 15;
26 Process comm: migration/0;
27 Policy: 1;
28 Prio: 0
29 Static_prio: 120;
30 Normal_prio: 0;
31 Rt_prio: 99;
32 Delay: 37071375;
33 Calls: 291;
34 Time of last run: 496177064338;
```

Значение поля `rt_prio` для данного процесса равняется 99, `normal_prio` = 0, а `policy` = 1, что означает, что при планировании использовался алгоритм `SCHED_FIFO`, что означает, что данный процесс является задачей реального времени.

Выводы

В данном разделе описаны технические характеристики машины, на которой проводились исследования, а также представлена демонстрация реализованного ПО.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы было разработано ПО, отвечающее поставленной задаче: предоставление пользователю возможности получения информации о планировании процессов в системе.

Для достижения поставленной задачи был разработан загружаемый модуль ядра, осуществляющий доступ к структурам ядра (`struct task_struct`, `struct sched_info`), хранящим соответствующую информацию и сохраняющий ее в файл VFS `proc`. Для получения данных в режиме пользователя в динамическом виде была реализована вспомогательная программа. Было показано, что процессы воспроизведения видеофайла и игры не являются задачами реального времени. Также рассмотрен процесс `migration`, который относится к задачам реального времени.

Т.к. вычислительные системы применяются в самых различных областях деятельности, каждая из которых обладает определенным набором требований, то ни один планировщик не может идеально подходить каждой системе. [6] Целесообразно выбирать алгоритмы планирования для каждой вычислительной системы индивидуально.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Вахалия Ю.* UNIX изнутри. — Питер, 2003. — С. 186.
2. Документация к UBUNTU scheduler. — Дата обновления: 13.02.2023. — URL: <https://ru.manpages.org/sched/7>.
3. C99 standart ISO/IEC 9899:TC3. — Дата обновления: 03.02.2023. — URL: <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>.
4. GCC, the GNU Compiler Collection. — Дата обновления: 03.02.2023. — URL: <https://gcc.gnu.org/>.
5. Visual Studio Code documentation. — Дата обновления: 03.02.2023. — URL: <https://code.visualstudio.com/docs>.
6. *Таненбаум Х.* Современные операционные системы. — Питер, 2015.

ПРИЛОЖЕНИЕ А

Исходный код

В листинге А.1 представлен исходный код загружаемого модуля ядра.

Листинг А.1 – Исходный код загружаемого модуля ядра

```
1  #include <linux/delay.h>
2  #include <linux/init.h>
3  #include <linux/init_task.h>
4  #include <linux/kernel.h>
5  #include <linux/module.h>
6  #include <linux/proc_fs.h>
7  #include <linux/kthread.h>
8
9  #define PROC_FS_NAME "bmstu_coursework"
10 #define DELIMITER "+++:"
11
12 #define TEMP_BUF_SIZE 1024
13
14 MODULE_LICENSE("GPL");
15 MODULE_AUTHOR("Sirotkina Polina");
16 MODULE_DESCRIPTION("Process planning monitoring");
17
18 #define REPEAT 5
19 #define DELAY_MS 10 * 1000
20
21 static struct proc_dir_entry *proc_file;
22
23 static struct task_struct *kthread;
24
25 #define BUF_SIZE 362144
26 static char out_str[BUF_SIZE] = { 0 };
27
28 static int check_buf(char *str_1, char *str_2, int max_len)
29 {
30     if ((strlen(str_1) + strlen(str_2)) >= max_len)
31     {
32         printk(KERN_ERR "%s check_buf failed.\n", DELIMITER);
33         return -ENOMEM;
34     }
35     return 0;
36 }
37
38 static int print_info(void *arg)
39 {
40     printk(KERN_INFO "%s print_info() called.\n", DELIMITER);
41 }
```

```

42     struct task_struct *task;
43
44     int i;
45     char cur_buf[TEMP_BUF_SIZE];
46
47     for (i = 0; i < REPEAT; i++)
48     {
49         task = &init_task;
50
51         memset(cur_buf, 0, TEMP_BUF_SIZE);
52         snprintf(cur_buf, TEMP_BUF_SIZE,
53                  "===== ITERATION #%d =====\n",
54                  i+1);
55
56         check_buf(cur_buf, out_str, BUF_SIZE);
57         strcat(out_str, cur_buf);
58
59         for_each_process(task) {
60             memset(cur_buf, 0, TEMP_BUF_SIZE);
61             snprintf(cur_buf, TEMP_BUF_SIZE,
62                      "PID: %d;\n"
63                      "Process comm: %s;\n"
64                      "Prio: %d;\n"
65                      "Policy: %d;\n"
66                      "Static_prio: %d;\n"
67                      "Normal_prio: %d;\n"
68                      "Rt_prio: %d;\n"
69                      "Delay: %lld;\n"
70                      "Calls: %ld;\n"
71                      "Time of last run: %lld;\n"
72                      "Last queued: %lld;\n\n\n",
73                      task->pid, task->comm,
74                      task->policy,
75                      task->prio, task->static_prio,
76                      task->normal_prio,
77                      task->rt_priority,
78                      task->sched_info.run_delay,
79                      task->sched_info.pcount,
80                      task->sched_info.last_arrival,
81                      task->sched_info.last_queued);
82
83             check_buf(cur_buf, out_str, BUF_SIZE);
84             strcat(out_str, cur_buf);
85         }
86         mdelay(DELAY_MS);
87     }
88
89     return 0;

```



```

90 }
91
92 static int my_open(struct inode *inode, struct file *file)
93 {
94     printk(KERN_INFO "%s my_open called.\n", DELIMITER);
95
96     if (!try_module_get(THIS_MODULE))
97     {
98         printk(KERN_INFO "%s error while try_module_get().\n", DELIMITER);
99         return -EFAULT;
100     }
101
102     return 0;
103 }
104
105 static ssize_t my_read(struct file *file, char __user *buf, size_t count,
106                       loff_t *offp)
107 {
108     ssize_t out_str_len = strlen(out_str);
109
110     printk(KERN_INFO "%s my_read called.\n", DELIMITER);
111
112     if (copy_to_user(buf, out_str, strlen(out_str)))
113     {
114         printk(KERN_ERR "%s error while copy_to_user().\n", DELIMITER);
115         return -EFAULT;
116     }
117
118     memset(out_str, 0, BUF_SIZE);
119
120     return out_str_len;
121 }
122
123 static ssize_t my_write(struct file *file, const char __user *buf,
124                        size_t len, loff_t *offp)
125 {
126     printk(KERN_INFO "%s my_write called.\n", DELIMITER);
127     return 0;
128 }
129
130 static int my_release(struct inode *inode, struct file *file)
131 {
132     printk(KERN_INFO "%s my_release called.\n", DELIMITER);
133     module_put(THIS_MODULE);
134     return 0;
135 }
136
137 static struct proc_ops my_ops = {

```

```

138     proc_read: my_read,
139     proc_write: my_write,
140     proc_open: my_open,
141     proc_release: my_release
142 };
143
144 static int __init md_init(void)
145 {
146     proc_file = proc_create(PROC_FS_NAME, 0666, NULL, &my_ops);
147
148     if (!proc_file)
149     {
150         printk(KERN_ERR "%s error while proc_create().\n", DELIMITER);
151
152         return -EFAULT;
153     }
154
155     kthread = kthread_run(print_info, NULL, "print_info_thread");
156     if (IS_ERR(kthread))
157     {
158         printk(KERN_ERR "%s error while kthread_run().\n", DELIMITER);
159
160         return -EFAULT;
161     }
162
163     printk(KERN_INFO "%s module loaded.\n", DELIMITER);
164
165     return 0;
166 }
167
168 static void __exit md_exit(void)
169 {
170     remove_proc_entry(PROC_FS_NAME, NULL);
171     printk(KERN_INFO "%s Module removed. Monitoring stopped.\n",
172     DELIMITER);
173 }
174
175 module_init(md_init);
176 module_exit(md_exit);

```

В листинге А.2 представлен исходный код вспомогательной программы, получающей информацию о планировании процессов из файла `proc` и сохраняющей результат в текстовый файл в пространстве ядра.

Листинг А.2 – Функция получения информации в режиме пользователя

```

1 #include <linux/sched/types.h>
2 #include <stdio.h>

```

```

3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/syscall.h>
6 #include <unistd.h>
7 #include <errno.h>
8
9 #define BUF_SIZE 32768
10
11 #define REPEAT 5
12 #define DELAY 10
13
14 int main(int argc, char *argv[])
15 {
16     FILE *proc_file = NULL;
17     FILE *out_file = fopen("out.txt", "w");
18
19     char buf[BUF_SIZE] = {'\0'};
20
21     for (int i = 0; i < REPEAT; i++)
22     {
23         proc_file = popen("cat /proc/bmstu_coursework", "r");
24
25         if (proc_file == NULL)
26         {
27             printf("Error while popen.");
28             return EXIT_FAILURE;
29         }
30
31         fgets(buf, sizeof(buf), proc_file);
32         fprintf(out_file, "%s", buf);
33
34         while (fgets(buf, sizeof(buf), proc_file) != NULL)
35         {
36             fprintf(out_file, "%s", buf);
37         }
38
39         pclose(proc_file);
40         sleep(DELAY);
41     }
42
43     fclose(out_file);
44
45     return EXIT_SUCCESS;
46 }

```