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 КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Мониторинг информации о процессах ОС Linux

Студент ИУ7-71Б _____ **Е. А. Варламова**
(Группа) (Подпись, дата) (И.О.Фамилия)

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

2022 г.

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

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
_____ И. В. Рудаков
(И.О.Фамилия)
« _____ » _____ 20 ____ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине Операционные системы

Студент группы ИУ7-71Б

Варламова Екатерина Алексеевна
(Фамилия, имя, отчество)

Тема курсового проекта Мониторинг информации о процессах ОС Linux

Направленность КП (учебный, исследовательский, практический, производственный, др.)
учебная

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание Разработать загружаемый модуль ядра Linux, предоставляющий информацию о процессах в системе за некоторый промежуток времени: их приоритетах, состояниях, времени выполнения, а также исполняющем ядре процессора.

Оформление курсового проекта:

Расчетно-пояснительная записка на 25-35 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
На защиту работы должна быть представлена презентация, состоящая из 10-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, интерфейс, результаты проведенных исследований.

Дата выдачи задания «1» сентября 2022 г.

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

Студент _____ Е.А. Варламова
(Подпись, дата) (И.О.Фамилия)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Планирование задач на исполнение является одной из важнейших задач операционной системы. В операционной системе Linux для организации планирования и выбора следующей задачи из очереди к ядру процессора используются значения приоритетов и время выполнения процессов.

Целью данного курсового проекта является мониторинг приоритетов, исполняющего ядра процессора и времени выполнения процессов в операционной системе Linux.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу по курсу «Операционные Системы» необходимо разработать загружаемый модуль ядра Linux, предоставляющий информацию о процессах в системе за некоторый промежуток времени: их приоритетах, состояниях, времени выполнения, а также исполняющем ядре процессора. Для этого необходимо:

1. проанализировать работу планировщика Linux;
2. проанализировать структуры ядра, содержащие необходимую информацию;
3. проанализировать и выбрать методы передачи информации из модуля ядра в пространство пользователя;
4. разработать алгоритмы, используемые в программном обеспечении;
5. провести исследование с помощью разработанного программного обеспечения, планируются ли процессы проигрывания аудио- и видеофайлов, а также игровые и интерактивные процессы в ОС Linux как процессы реального времени..

В результате загрузки разработанного модуля будет создан файл в файловой системе proc, содержащий эту информацию.

1.2 Анализ работы планировщика

Основной функцией планировщика является выбор задач на исполнение процессором. Каждая задача имеет свой алгоритм планирования и приоритет планирования sched_priority. Планировщик принимает решения на основе алгоритма планирования и приоритета sched_priority всех задач в системе [1]. Алгоритмы планирования можно разделить на 2 типа: обычные и алгоритмы реального времени.

Для задач, запланированных в соответствии с одним из обычных алгоритмов планирования, sched_priority равен 0. Процессы, запланированные в соответствии с одним из алгоритмов реального времени, имеют значение sched_priority в диапазоне от 1 (низкий) до 99 (высокий).

Планировщик поддерживает список задач для каждого возможного значения sched_priority. Чтобы определить, какую задачу выполнить следующей, планировщик ищет непустой список с наивысшим приоритетом sched_priority и выбирает задачу из головы этого списка. Алгоритм планирования определяет, куда задача будет вставлена в список задач с равным приоритетом sched_priority и как она будет перемещаться внутри этого списка.

Планировщик поддерживает следующие 6 алгоритмов планирования [2].

- Обычные алгоритмы планирования. Для их реализации с версии 2.6.23 используется CFS («Completely Fair Scheduler» – «Полностью честный планировщик»), в основе которого упорядоченное красно-чёрное дерево, где все выполняемые задачи сортируются по ключу p->se.vruntime (взвешенное время использования процессора – время использования процессора с учётом динамического приоритета). При планировании выбирается «крайняя левая» задача из этого дерева и выполняется на процессоре до тех пор, пока не найдётся задача в дереве, располагающаяся «левее», чем текущая.
  - SCHED_NORMAL (SCHED_OTHER): используется для обычных задач;
  - SCHED_BATCH: алгоритм, при котором вытеснение происходит реже, чем при SCHED_NORMAL, за счёт чего достигается более продолжительное выполнение одной задачи и лучшее использование кэша, однако при этом ухудшается интерактивность.
  - SCHED_IDLE: может быть использован для низкоприоритетных задач.
- Алгоритмы реального времени.
  - SCHED_FIFO: задачи выбираются из соответствующей приоритету sched_priority очереди и выполняются до тех пор, пока не будут заблокированы или вытеснены другой более приоритетной

задачей; при переходе в состояние runnable или при изменении приоритета задача помещается в конец очереди.

- SCHED_RR: усовершенствование SCHED_FIFO, при котором каждой задаче разрешено выполняться только в течение некоторого временного интервала; после его истечения она будет помещена в конец списка своего приоритета.
- SCHED_DEADLINE: доступна с версии 3.14; основана на алгоритме EDF; в основе алгоритма планирования лежит период P, соответствующий объявлению ядру о том, что Q единиц времени требуется для этой задачи каждые P единиц времени на любом ядре процессора.

1.3 Анализ структур ядра, предоставляющих информацию о процессах

1.3.1 Структура task_struct

Структура task_struct в ядре Linux описывает каждый процесс в системе. [3]. Важные для данной работы поля структуры приведены в листинге 1.1.

```
struct task_struct {
...
  unsigned int      __state;
  unsigned int      cpu;
  int       prio;
  int       static_prio;
  int       normal_prio;
  unsigned int      rt_priority;
  struct sched_entity   se;
    unsigned int      policy;
    struct sched_info   sched_info;
  pid_t       pid;
  u64       utime;
  u64       stime;
 ...
 }
```

Листинг 1.1: Структура task_strcut.

Идентификатор процесса pid Каждый процесс в операционной системе имеет свой уникальный идентификатор, по которому можно получить информацию об этом процессе, а также направить ему управляющий сигнал или завершить его.

Состояние процесса __state каждый процесс в операционной системе в каждый момент времени находится в некотором состоянии. Поле __state описывает это состояние. Соответствие значений этого поля состояниям показано в листинге 1.2.

```
#define TASK_RUNNING      0x00000000
#define TASK_INTERRUPTIBLE    0x00000001
#define TASK_UNINTERRUPTIBLE    0x00000002
#define __TASK_STOPPED      0x00000004
#define __TASK_TRACED     0x00000008
/* Used in tsk->exit_state: */
#define EXIT_DEAD     0x00000010
#define EXIT_ZOMBIE     0x00000020
#define EXIT_TRACE      (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED     0x00000040
#define TASK_DEAD     0x00000080
#define TASK_WAKEKILL     0x00000100
#define TASK_WAKING     0x00000200
#define TASK_NOLOAD     0x00000400
#define TASK_NEW      0x00000800
#define TASK_RTLOCK_WAIT    0x00001000
#define TASK_FREEZABLE      0x00002000
#define __TASK_FREEZABLE_UNSAFE        (0x00004000 * IS_ENABLED(
   CONFIG_LOCKDEP))
#define TASK_FROZEN     0x00008000
#define TASK_STATE_MAX      0x00010000
```

Листинг 1.2: Соответствие значений поля __state состояниям процесса

static_prio – статический приоритет процесса ($[100; 139]$) не изменяется ядром при работе планировщика, однако может быть изменено пользователем с помощью макроса NICE_TO_PRIO. При создании процесса значение статического приоритета либо наследуется от родительского процесса, либо при наличии флага reset_on_fork выставляется по умолчанию, то есть становится равным 120 [4].

rt_priority – приоритет процесса реального времени ($[0; 99]$). При назначении алгоритма планирования rt_priority приравнивается к sched_priority (п. 1.2). Значение этого приоритета определяет, является ли процесс задачей реального времени.

normal_prio – нормальный приоритет ([-1; 139]) для обычных процессов равняется значению статического приоритета static_prio; для процессов реального времени равняется значению, вычисленному с использованием максимального значения приоритета и приоритета процесса реального времени. Для обычных процессов значение в диапазоне $[100, 139]$, для процессов с алгоритмом SCHED_FIFO и SCHED_RR – в диапазоне $[0; 99]$, для процессов с алгоритмом SCHED_DEADLINE равно -1.

prio – значение приоритета, которое использует планировщик. Для процессов реального времени равен rt_priority (т.е. находится в диапазоне $[0; 99]$), для обычных процессов равен normal_prio (т.е. static_prio, находится в диапазоне $[100; 139]$. Чем ниже значение prio, тем выше приоритет процесса. Также на основании анализа значения этого приоритета функция rt_task определяет принадлежность процесса к процессам реального времени.

policy – принимает значения от 0 до 6 (4 пока зарезервировано) и указывает на алгоритм, в соответствии с которым запланирована задача. Соответствие значений поля policy алгоритмам показано в листинге 1.3.

```
include/uapi/linux/sched.h

#define SCHED_NORMAL    0
#define SCHED_FIFO    1
#define SCHED_RR    2
#define SCHED_BATCH   3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE    5
#define SCHED_DEADLINE    6
```

Листинг 1.3: Соответствие значений поля policy алгоритмам планирования

utime – это время, проведенное в режиме пользователя и затраченное на запуск команд. Данное значение включает в себя только время, затраченное

центральным процессором, и не включает в себя время, проведенное процессом в очереди на исполнение.

stime – это время процессора, затраченное на выполнение системных вызовов при исполнении процесса.

cpu – это номер процессора, на котором исполняется задача.

1.3.2 Структура sched_info

sched_info – структура, которая предоставляет информацию о планировании процесса [3]. Данная структура представлена в листинге 1.4.

```
struct sched_info {
  /* Cumulative counters: */

  /* # of times we have run on this CPU: */
  unsigned long     pcount;

  /* Time spent waiting on a runqueue: */
  unsigned long long    run_delay;

  /* Timestamps: */

  /* When did we last run on a CPU? */
  unsigned long long    last_arrival;

  /* When were we last queued to run? */
  unsigned long long    last_queued;
};
```

Листинг 1.4: Структура sched_info.

- количество запусков процесса на исполнение центральным процессором (поле pcount);
- количество времени, проведенного в ожидании на исполнение (поле run_delay);
- время последнего запуска процесса на исполнение центральным процессором (поле last_arrival);

- время последнего добавления процесса в очередь на исполнение (поле last_queued).

1.3.3 Структура sched_entity

Данная структура описывает процесс как единицу планирования [3]. Наиболее важными для данной работы являются поля, представленные в листинге 1.5.

```
struct sched_entity {
  /* For load-balancing: */
...
  u64        exec_start;
  u64        sum_exec_runtime;
  u64        vruntime;
...
};
```

Листинг 1.5: Структура sched_entity.

Для обычных процессов поле vruntime является ключом в красно-чёрном дереве CFS (п. 1.2) и определяется как взвешенная разница между временем последнего обновления статистики по времени текущей задачи (поле exec_start) и временем в настоящий момент.

Поле sum_exec_runtime является суммарным временем выполнения на процессоре текущей задачи для всех алгоритмов планирования.

Важно отметить, что существует функция, которая корректирует значения utime и stime таким образом, чтобы сумма равнялась sum_exec_runtime. Она вызывается при обращении к /proc/pid/stat.

1.4 Передача данных из пространства ядра в пространство пользователя

Файловая система /proc представляет собой интерфейс ядра, который позволяет получать информацию о процессах и ресурсах, которые они используют. При этом используется стандартный интерфейс файловой системы и системных вызовов. Структура proc_ops используется для определения

обратных вызовов чтения и записи. Важные для данной работы поля структуры представлены в листинге 1.6 [3].

```
struct proc_ops {
...
  int (*proc_open)(struct inode *, struct file *);
  ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
  int (*proc_release)(struct inode *, struct file *);
 ...
};
```

Листинг 1.6: Структура proc_ops.

Кроме того, из-за разных уровней привилегий пространства пользователя и пространства ядра для копирования блоков данных из пространства ядра в пространство пользователя необходимо использовать специальную функцию – copy_to_user [3], определённую в файле include/linux/uaccess.h.

Выводы

В результате анализа кода ядра были определены структуры, содержащие необходимую информацию о процессах в системе: task_struct, sched_info и sched_entity. Кроме того, был определён механизм передачи данных из пространства ядра в пространство пользователя.

2 Конструкторский раздел

2.1 Диаграмма состояний (IDEF0)

На рисунках 2.1 и 2.2 показаны соответственно нулевой и первый уровни диаграммы IDEF0, отображающие процесс мониторинга информации о процессах.

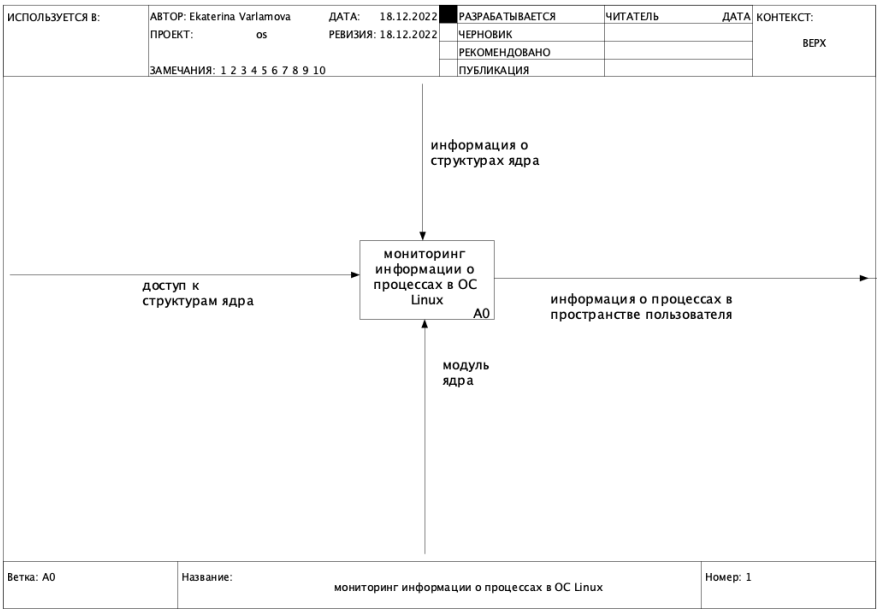

Рис. 2.1: IDEF0 нулевого уровня.

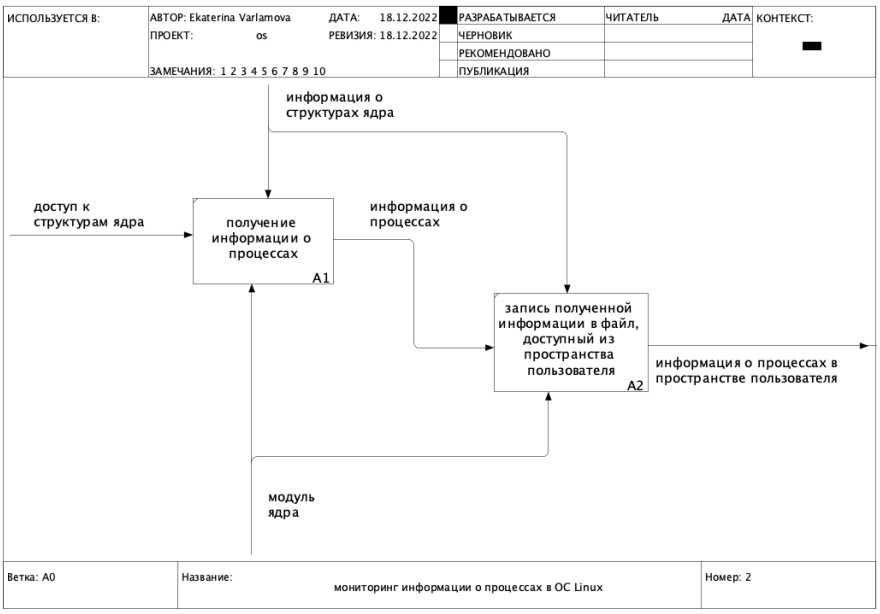

Рис. 2.2: IDEF0 первого уровня.

2.2 Алгоритмы для мониторинга информации о процессах

Алгоритмы модуля ядра

На рисунке 2.4 представлен алгоритм вывода информации о процессах в файл в /proc. Данный алгоритм должен быть выполнен в момент чтения файла /proc. В алгоритме также предполагается, что файл в /proc уже создан.

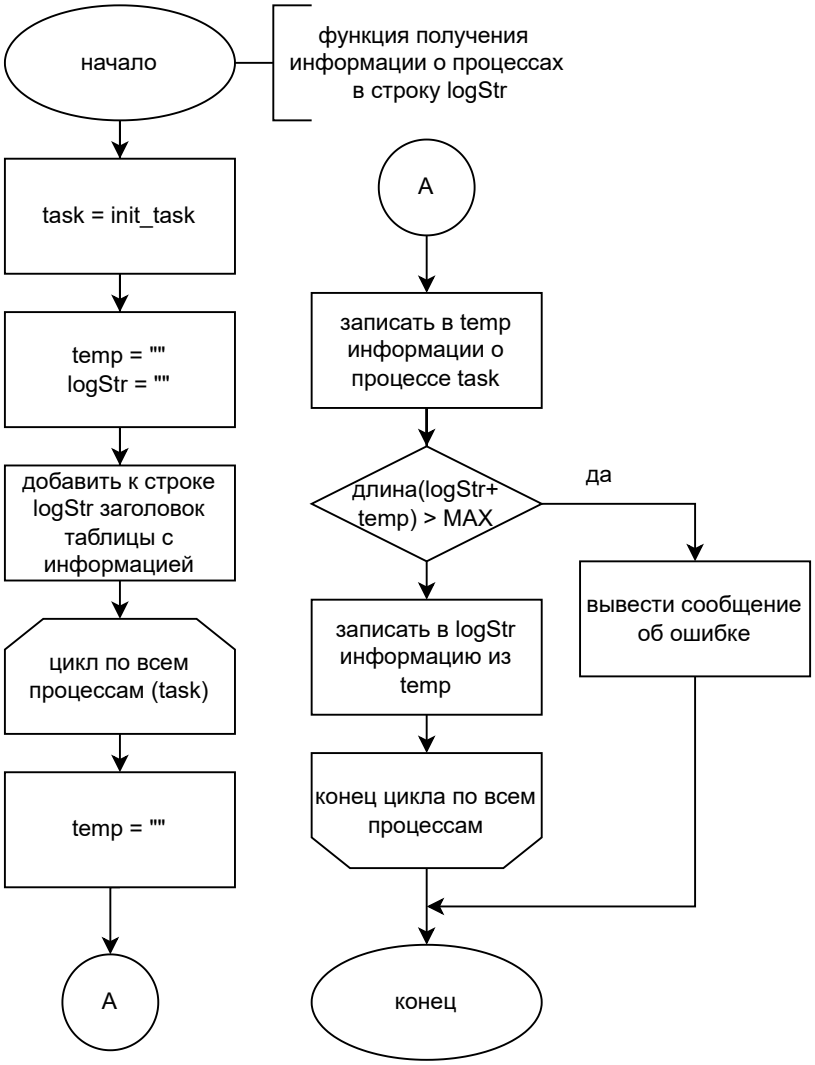

Рис. 2.3: Алгоритм вывода информации о процессах в файл в /proc

На рисунке 2.4 представлен алгоритм получения информации о процессах из структур ядра.

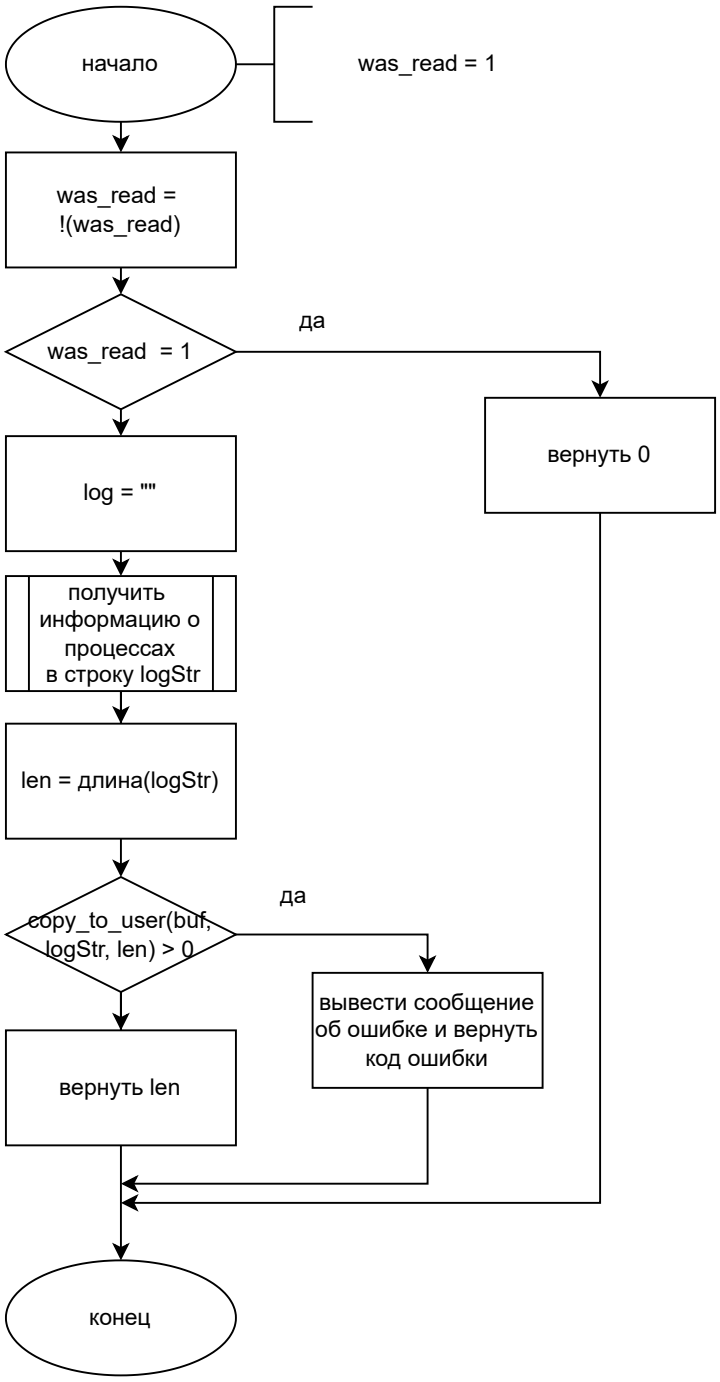

Рис. 2.4: Алгоритм получения информации о процессах из структур ядра

Алгоритм получения информации из файла в procfs в пространстве пользователя

Для исследования процессов в динамике был разработан алгоритм получения информации из /proc в течение заданного количества секунд с интервалом в одну секунду. Алгоритм представлен на рисунке 2.5

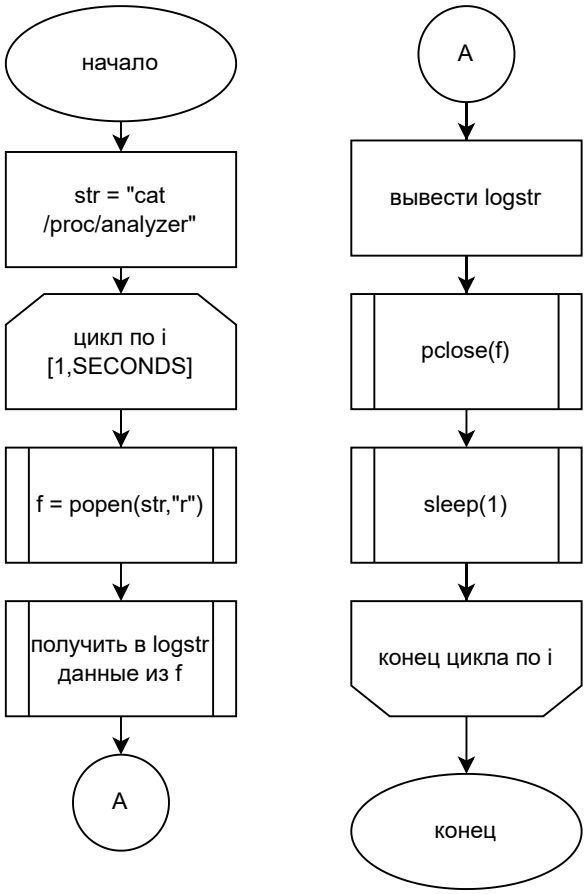

Рис. 2.5: Алгоритм получения информации из /proc

2.3 Структура разработанного ПО

На рисунке 2.6 представлена структура разработанного ПО.

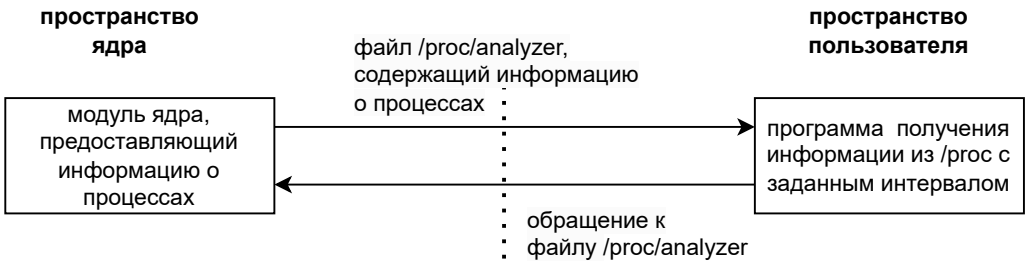

Рис. 2.6: Структура ПО

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для написания программного кода использовался язык программирования C [5], так как в реализации использовались структуры ядра Linux, а исходный код ядра написан на C.

В качестве среды программирования использовалась среда CLion, так как она предоставляет широкий выбор плагинов для работы с кодом на языке C.

3.2 Реализация алгоритмов мониторинга информации о процессах

В листинге 3.1 представлен код алгоритма вывода информации о процессах в файл /proc.

```
static char log[LOG_SIZE] = { 0 };
int was_read = 1;
static ssize_t my_read(struct file *filep, char __user *buf, size_t count,
    loff_t *offp)
{
    was_read = !was_read;
    if (was_read)
        return 0;

    memset(log, 0, LOG_SIZE);
    print_tasks();
    ssize_t logLen = strlen(log);
    printk(KERN_INFO "read called\n");

    if (copy_to_user(buf, log, logLen))
    {
        printk(KERN_ERR "copy_to_user error\n");
        return -EFAULT;
    }

    return logLen;
}
```

Листинг 3.1: Реализация алгоритма вывода информации о процессах

В листинге 3.2 представлен код алгоритма получения информации о процессах из структур ядра.

```
#define TEMP_STRING_SIZE 512
#define LOG_SIZE 256 * 1024
static char log[LOG_SIZE] = { 0 };
static struct proc_dir_entry *proc_file;
void print_tasks(void)
{
    struct task_struct *task;
    char temp[TEMP_STRING_SIZE];
    task = &init_task;
    memset(temp, 0, TEMP_STRING_SIZE);
    snprintf(temp, TEMP_STRING_SIZE,
             "%-5s %15s %6s %4s %5s %5s %5s %3s %6s %15s %15s %16s %16s %16s
    %6s %15s %18s %11s\n",
             "PID","name", "state", "prio", "sprio",
             "nprio", "rprio", "cpu", "policy",
             "utime", "stime", "exec_start","sum_exec_runtime",
             "vruntime", "pcount", "run_delay", "last_arrival", "last_queued
   ");
    strcat(log, temp);
    for_each_process(task)
    {
        memset(temp, 0, TEMP_STRING_SIZE);
        snprintf(temp, TEMP_STRING_SIZE,
                 "%-5d %15s %6d %4d %5d %5d %5d %3d %6d %15lld %15lld %16lld
    %16lld %16lld %6lu %15llu %18llu %11llu\n",
                 task->pid, task->comm, task->__state,
                 task->prio, task->static_prio, task->normal_prio, task->
   rt_priority,
                 task->cpu, task->policy,
                 task->utime, task->stime,
                 task->se.exec_start, task->se.sum_exec_runtime, task->se.
   vruntime,
                 task->sched_info.pcount, task->sched_info.run_delay, task->
   sched_info.last_arrival, task->sched_info.last_queued);

        if (strlen(temp) + strlen(log) < LOG_SIZE)
            strcat(log, temp);
        else
            printk(KERN_ERR "max log size was exceeded!\n");
    }
}
```

Листинг 3.2: Реализация алгоритма получения информации о процессах из структур ядра

В листинге 3.3 представлен код алгоритма получения информации о про-

цессах из файла в /proc в пространстве пользователя.

```
...
#include <...>
#define SIZE_OF_LOG 8192
#define ERROR_COMMAND_EXEC 1
#define TIMES 100
#define DELAY 1
int main(int argc, char *argv[])
{
    FILE *filePointer = NULL;
    char log[SIZE_OF_LOG] = {'\0'};
    for (int i = 0; i < TIMES; i++)
    {
        filePointer = popen("cat /proc/analyzer", "r");
        if (filePointer == NULL)
        {
            printf("Error: can't execute cat for process analyzer");
            return ERROR_COMMAND_EXEC;
        }
        while (fgets(log, sizeof(log), filePointer) != NULL)
        {
            printf("%s", log);
            fgets(log, sizeof(log), filePointer);
            printf("%s", log);
        }
        pclose(filePointer);
        sleep(DELAY);
    }
    return EXIT_SUCCESS;
}
...
```

Листинг 3.3: Реализация алгоритма получения информации о процессах из файла в /proc в пространстве пользователя

В листинге 3.4 представлен полный код модуля ядра, предоставляющего файл в /proc для мониторинга информации о процессах .

```
#include <...>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Varlamova Ekaterina");
#define PROC_FS_NAME "analyzer"
#define TEMP_STRING_SIZE 512
#define LOG_SIZE 256 * 1024
static char log[LOG_SIZE] = { 0 };
static struct proc_dir_entry *proc_file;
int was_read = 1;
void print_tasks(void)
```

```
{
    struct task_struct *task;
    char temp[TEMP_STRING_SIZE];
    task = &init_task;
    memset(temp, 0, TEMP_STRING_SIZE);
    snprintf(temp, TEMP_STRING_SIZE,
             "%-5s %15s %6s %4s %5s %5s %5s %3s %6s %15s %15s %16s %16s %16s
    %6s %15s %18s %11s\n",
             "PID","name", "state", "prio", "sprio",
             "nprio", "rprio", "cpu", "policy",
             "utime", "stime", "exec_start","sum_exec_runtime",
             "vruntime", "pcount", "run_delay", "last_arrival", "last_queued
   ");
    strcat(log, temp);
    for_each_process(task)
    {
        memset(temp, 0, TEMP_STRING_SIZE);
        snprintf(temp, TEMP_STRING_SIZE,
                 "%-5d %15s %6d %4d %5d %5d %5d %3d %6d %15lld %15lld %16lld
    %16lld %16lld %6lu %15llu %18llu %11llu\n",
                 task->pid, task->comm, task->__state,
                 task->prio, task->static_prio, task->normal_prio, task->
   rt_priority,
                 task->cpu, task->policy,
                 task->utime, task->stime,
                 task->se.exec_start, task->se.sum_exec_runtime, task->se.
   vruntime,
                 task->sched_info.pcount, task->sched_info.run_delay, task->
   sched_info.last_arrival, task->sched_info.last_queued);
        if (strlen(temp) + strlen(log) < LOG_SIZE)
            strcat(log, temp);
        else
            printk(KERN_ERR "max log size was exceeded!\n");
    }
}
static ssize_t my_read(struct file *filep, char __user *buf, size_t count,
   loff_t *offp)
{
    was_read = !was_read;
    if (was_read)
        return 0;
    memset(log, 0, LOG_SIZE);
    print_tasks();
    ssize_t logLen = strlen(log);
    printk(KERN_INFO "read called\n");
    if (copy_to_user(buf, log, logLen))
    {
        printk(KERN_ERR "copy_to_user error\n");
```

```
        return -EFAULT;
    }
    return logLen;
}
static int my_open(struct inode *spInode, struct file *spFile)
{
    printk(KERN_INFO "open called\n");
    return 0;
}
static int my_release(struct inode *spInode, struct file *spFile)
{
    printk(KERN_INFO "release called\n");
    return 0;
}
static struct proc_ops ops = {
        proc_read : my_read,
        proc_open: my_open,
        proc_release: my_release
};
static int __init md_init(void)
{
    if (!(proc_file = proc_create(PROC_FS_NAME, 0666, NULL, &ops)))
    {
        printk(KERN_ERR "proc_create error\n");
        return -EFAULT;
    }

    printk(KERN_INFO "module loaded\n");

    return 0;
}
static void __exit md_exit(void)
{
    remove_proc_entry(PROC_FS_NAME, NULL);
    printk(KERN_INFO "module exited\n");
}
module_init(md_init);
module_exit(md_exit);
```

Листинг 3.4: Код модуля ядра

Выводы

В результате, был разработан модуль ядра Linux, реализующий мониторинг информации о процессах в соответствии с разработанными в конструк-

торском разделе алгоритмами, а также теоретическими сведениями о структурах ядра Linux, полученными в аналитическом разделе.

4 Исследовательский раздел

4.1 Условия исследований

Исследование проводилось на компьютере со следующими характеристиками:

- 8-ядерный процессор 11th Gen Intel Core i7-1165G7;
- операционная система Linux (дистрибутив Ubuntu 20.04.5 LTS, версия ядра Linux 5.15.0-56, архитектура x86-64);
- 16 Гб оперативной памяти.

Для динамического анализа информации о процессах была разработана программа, выводящая информацию из файла /proc/analyzer заданное количество раз с интервалом времени в 1 секунду.

Для получения идентификатора исследуемого процесса использовалась утилита lsof [2].

Для фильтрации вывода разработанной программы по идентификатору процесса использовались утилиты awk и grep [2].

4.2 Исследование процесса проигрывания аудиофайла

На рисунке 4.1 представлен вывод команд lsof и разработанной программы. Так как приоритет процесса равен 120, можно сделать вывод, что процесс проигрывания аудиофайла был запланирован как обычный процесс (не процесс реального времени).

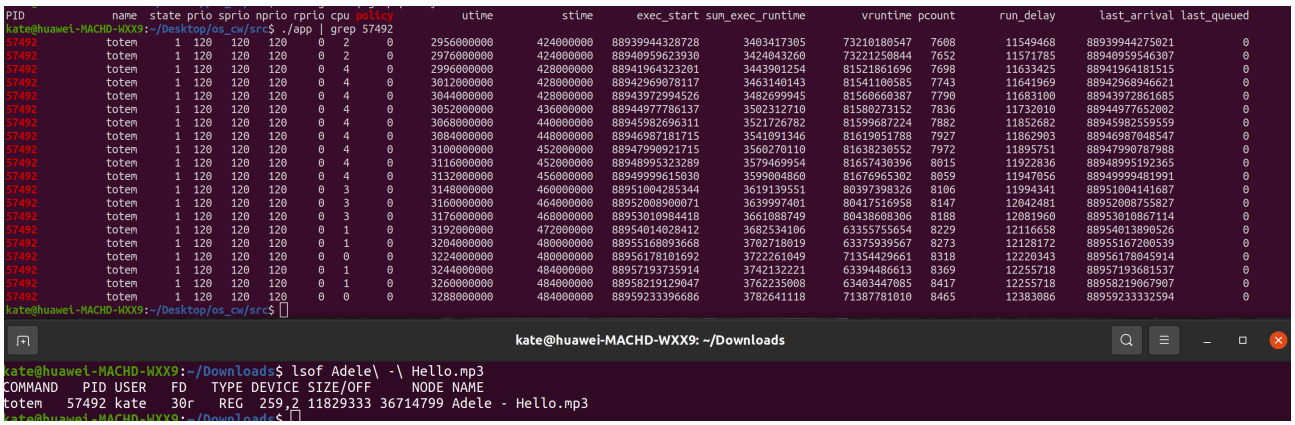

Рис. 4.1: Вывод программы при проигрывании аудиофайла

4.3 Исследование процесса проигрывания видеофайла

На рисунке 4.2 представлен вывод команды lsof и разработанной программы. Так как приоритет процесса равен 120, можно сделать вывод, что процесс проигрывания видеофайла был запланирован как обычный процесс (не процесс реального времени).

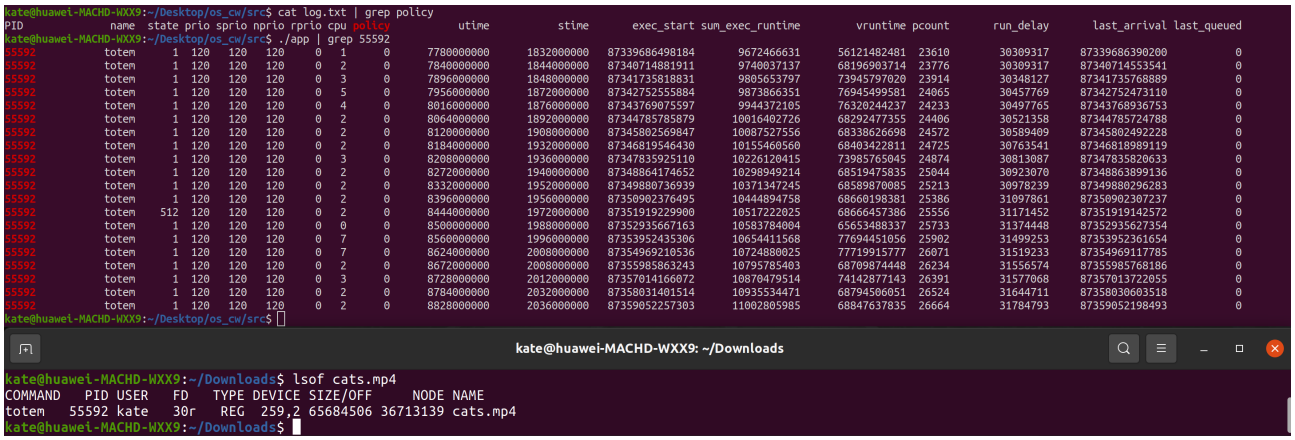

Рис. 4.2: Вывод программы при проигрывании видеофайла

4.4 Исследование процесса игры

На рисунке 4.3 представлен вывод разработанной программы во время игры. Так как приоритет игрового процесса равен 120, можно сделать вывод, что процесс игры был запланирован как обычный процесс (не процесс реального времени).

```
kate@huawei-MACHD-WXX9:~/Desktop/os_cw/src$ ./app | awk '$1 == 7693'
7693  Crab Game.x86_6      1  120   120   120     0   6      0     28816000000      2912000000      981512439902      31674956808      93344253810  76110     204137032      981512405703          0
7693  Crab Game.x86_6      1  120   120   120     0   6      0     29008000000      2952000000      982547419615      31900759295      93422685856  76637     206820682      982547388475          0
7693  Crab Game.x86_6      1  120   120   120     0   6      0     29148000000      2964000000      983562202528      32062038873      93487516493  77122     207576097      983562170097          0
7693  Crab Game.x86_6      1  120   120   120     0   2      0     29296000000      2980000000      984572124259      32233702157      161866102497 77583     208221057      984572085096          0
7693  Crab Game.x86_6      1  120   120   120     0   5      0     29440000000      2996000000      985600606988      32386211986      60925092649  78059     209023911      985600591463          0
7693  Crab Game.x86_6      1  120   120   120     0   2      0     29560000000      3008000000      986604721340      32523354083      162021183417 78532     209585780      986604696211          0
7693  Crab Game.x86_6      1  120   120   120     0   3      0     29684000000      3020000000      987617662825      32668214822      72060406807  79067     210587051      987617204174          0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     29804000000      3052000000      988625662270      32810116829      42608037570  79554     211786659      988624900824          0
7693  Crab Game.x86_6      1  120   120   120     0   7      0     29948000000      3080000000      989661831199      32980190631      85585222740  80094     212953648      989661806399          0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     30060000000      3100000000      990668444531      33120192125      42695821268  80595     213814932      990668426071          0
7693  Crab Game.x86_6      1  120   120   120     0   7      0     30180000000      3124000000      991675177508      33258007012      85681006843  81077     215440332      991674204704          0
7693  Crab Game.x86_6      0  120   120   120     0   4      0     30308000000      3140000000      992698865673      33406625948      104941481758 81512     215645048      992698865673          0
7693  Crab Game.x86_6      1  120   120   120     0   5      0     30420000000      3152000000      993700323279      33538487569      61179984297  81974     216543449      993699924298          0
7693  Crab Game.x86_6      1  120   120   120     0   1      0     30564000000      3168000000      994724866466      33696926647      143328582315 82440     217376993      994724840294          0
7693  Crab Game.x86_6      0  120   120   120     0   1      0     30708000000      3180000000      995739101207      33862174922      143394552704 82927     218419381      995739101207          0
7693  Crab Game.x86_6      1  120   120   120     0   4      0     30816000000      3200000000      996738672635      33984807342      105138351470 83393     219050115      996737924401          0
7693  Crab Game.x86_6      1  120   120   120     0   5      0     30944000000      3216000000      997756190360      34125524089      61373227488  83892     220643520      997755748318          0
7693  Crab Game.x86_6      0  120   120   120     0   3      0     31056000000      3220000000      998772805968      34251007807      72565167314  84407     221358433      998772805968          0
7693  Crab Game.x86_6      0  120   120   120     0   7      0     31188000000      3232000000      999790129166      34399320233      86154708289  84920     221823441      999790129166          0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     31312000000      3256000000      1000783942135     34558771779      43083027467  85419     222954955      1000783428618         0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     31452000000      3268000000      1001795814719     34725496622      43168743854  85959     225326683      1001795555411         0
7693  Crab Game.x86_6      1  120   120   120     0   1      0     31608000000      3284000000      1002815428427     34891964400      143702586005 86473     227299187      1002815411105         0
7693  Crab Game.x86_6      0  120   120   120     0   2      0     31748000000      3300000000      1003834209612     35047158287      162814118951 86958     228891613      1003834209612         0
7693  Crab Game.x86_6      1  120   120   120     0   4      0     31908000000      3312000000      1004839502825     35218846623      105517741660 87460     229288668      1004839486051         0
7693  Crab Game.x86_6      0  120   120   120     0   4      0     32028000000      3328000000      1005857393511     35363391388      105594058498 87962     230088516      1005857393511         0
7693  Crab Game.x86_6      1  120   120   120     0   4      0     32128000000      3340000000      1006851284728     35480819722      105632644080 88452     231056864      1006850907322         0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     32236000000      3348000000      1007867673002     35602677970      43446042566  88966     232271078      1007867272413         0
7693  Crab Game.x86_6      1  120   120   120     0   5      0     32344000000      3372000000      1008874086539     35711676950      61848570597  89464     233681222      1008874070938         0
7693  Crab Game.x86_6      1  120   120   120     0   3      0     32476000000      3384000000      1009901701777     35839122246      73044452639  89914     234192843      1009901683470         0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     32600000000      3412000000      1010912523111     35989251195      43574966734  90391     235389063      1010912054095         0
7693  Crab Game.x86_6      1  120   120   120     0   0      0     32708000000      3416000000      1011902478285     36106619163      43627610178  90892     236662490      1011902053559         0
7693  Crab Game.x86_6      1  120   120   120     0   6      0     32844000000      3424000000      1012924814839     36262230509      94964934952  91388     237275413      1012924793138         0
```

Рис. 4.3: Вывод программы во время игрового процесса

4.5 Исследование интерактивного процесса

Для проведения исследования дополнительно была разработана программа, которая читает введённые через терминал символы с помощью системного вызова read. Кроме того, был разработан модуль ядра, в котором регистрируется обработчик прерывания на событие на клавиатуре, в котором выводится приоритет исследуемого интерактивного процесса.

На рисунке 4.4 представлен вывод программы интерактивного процесса. Так как приоритет процесса равен 120, можно сделать вывод, что интерактивный процесс был запланирован как обычный процесс (не процесс реального времени).

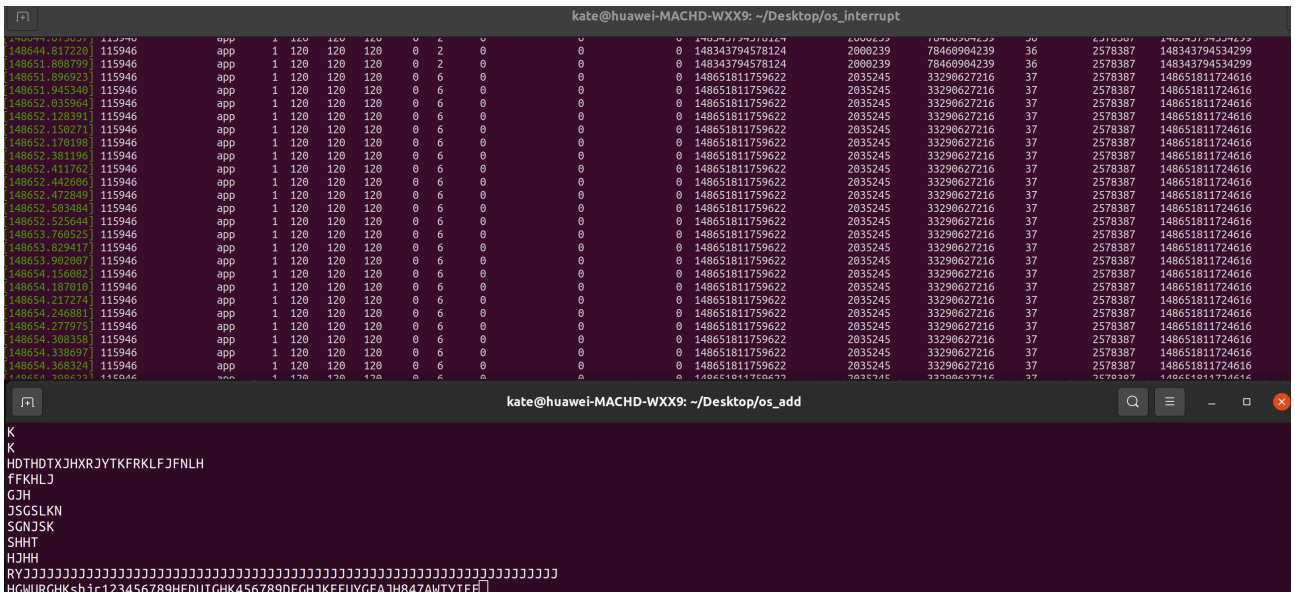

Рис. 4.4: Вывод программы интерактивного процесса

4.6 Исследование ситуации инверсии приоритетов

Инверсия приоритетов – ситуация в системе, при которой процесс с высоким приоритетом вынужден ждать выполнение процесса с низким приоритетом. Пусть в системе существуют такие 3 процесса, что:

- процесс A с низким приоритетом;
- процесс C с высоким приоритетом;
- процесс B со «средним» приориетом.

Рассмотрим ситуацию, когда процесс A успевает захватить ресурс (мьютекс) раньше процесса C. При этом процесс C после захвата мьютекса может быть вытеснен процессом B из-за более низкого приоритета. Таким образом, процесс A не может выполняться из-за захвата мьютекса процессом C, а процесс C не может выполняться (и отпустить ресурс), так как вытесняется процессом B. Следовательно, высокоприоритетный процесс C ждет выполнение процесса B со средним приоритетом.

Такая ситуация в ядре решается **временным увеличением** приоритета процесса A до приоритета процесса C до освобождения ресурса. Это называется наследованием приоритетов (priority inheritance).

Для проведения исследования:

- был использован код, который создаёт по 3 потока на каждое ядро процессора (с приоритетами 96, 97, 98) и с помощью захвата мьютекса имитирует ситуацию инверсии приоритетов;
- был изменён модуль ядра таким образом, чтобы информация выводилась в разрезе потоков, а не процессов;
- был изменён код пространства пользователя, который периодически обращается к файлу в /proc, таким образом, чтобы информация выводилась в бесконечном цикле без задержек (чтобы не пропустить изменение приоритетов).

На рисунке 4.5 представлен вывод программы пространства пользователя. На рисунке видно, что процесс 15407 в первой итерации вывода имеет normal_prio = 98 и prio = 96, а во второй итерации normal_prio = 98 и prio = 98. Таким образом, низкоприоритетному процессу был временно повышен приоритет до высокоприоритетного для выхода из ситуации инверсии приоритетов.

PID	name	state	prio	sprio	nprio	rprio	cpu	policy	utime	stime	exec_start	sum_exec_runtime	vruntime	pcount	run_delay	last_arrival	last_queued
15403	stress_pi_test	1	95	120	95	4	0	1	4000000	0	2613873262372	2063217	0	5	7190	2613873262281	0
15404	stress_pi_test	0	98	120	98	1	1	1	68000000	1000000000	2614274427079	802874532	0	487965	1098081110	2614274425917	2614274427079
15405	stress_pi_test	0	97	120	97	2	1	1	60000000	372000000	2614274425917	431806752	0	390364	617253011	2614274424756	2614274429691
15406	stress_pi_test	1	96	120	96	3	1	1	72000000	280000000	2614274432290	616475111	0	487956	224110614	2614274430760	0
15407	stress_pi_test	0	96	120	98	1	2	1	124000000	900000000	2614274434424	805168466	0	476679	1095790371	2614274434424	0
15408	stress_pi_test	0	97	120	97	2	2	1	72000000	388000000	2614274431774	429833688	0	381336	565140377	2614274430635	2614274435338
15409	stress_pi_test	1	96	120	96	3	2	1	80000000	280000000	2614274438317	616128544	0	476671	176733390	2614274436518	0
15410	stress_pi_test	0	98	120	98	1	3	1	92000000	960000000	2614274441309	804640297	0	470960	1096498046	2614274439417	2614274441309
15411	stress_pi_test	0	97	120	97	2	3	1	40000000	392000000	2614274443592	424671017	0	376766	616983966	2614274442390	2614274445405
15412	stress_pi_test	512	96	120	96	3	3	1	68000000	300000000	2614274446464	621908123	0	470958	227974298	2614274445405	2614274449293
15413	stress_pi_test	0	98	120	98	1	4	1	132000000	896000000	2614274451485	802463502	0	707080	1098573894	2614274449636	2614274451485
15414	stress_pi_test	0	97	120	97	2	4	1	56000000	328000000	2614274456250	432666993	0	565658	617826718	2614274452583	2614274455253
15415	stress_pi_test	1	96	120	96	3	4	1	80000000	360000000	2614274459890	615989618	0	707074	225137122	2614274458912	0
15416	stress_pi_test	0	98	120	98	1	5	1	104000000	984000000	2614274462220	803094020	0	484911	1097916501	2614274460376	2614274462220
15417	stress_pi_test	0	97	120	97	2	5	1	52000000	344000000	2614274467430	431420121	0	387924	617805659	2614274467430	0
15418	stress_pi_test	1	96	120	96	3	5	1	48000000	320000000	2614274471398	616653431	0	484905	221713225	2614274469731	0
15419	stress_pi_test	0	98	120	98	1	6	1	84000000	908000000	2614274476609	805004082	0	469225	1095964220	2614274475822	0
15420	stress_pi_test	512	97	120	97	2	6	1	84000000	428000000	2614274479646	432268876	0	375377	614156632	2614274478556	2614274481498
15421	stress_pi_test	0	96	120	96	3	6	1	72000000	276000000	2614274485788	613973147	0	469223	174870347	2614274485788	0
15422	stress_pi_test	0	98	120	98	1	7	1	84000000	952000000	2614274490140	809344734	0	470010	1091848729	2614274487961	2614274490140
15423	stress_pi_test	1	97	120	97	2	7	1	40000000	396000000	2614274496773	433885030	0	376006	611738193	2614274495734	0
15424	stress_pi_test	1	96	120	96	3	7	1	100000000	280000000	2614274499846	608063116	0	470008	177082656	2614274498785	0
15424	stress_pi_test	1	96	120	96	3	7	1	100000000	280000000	2614274499846	608063116	0	470008	177082656	2614274498785	0
PID	name	state	prio	sprio	nprio	rprio	cpu	policy	utime	stime	exec_start	sum_exec_runtime	vruntime	pcount	run_delay	last_arrival	last_queued
15403	stress_pi_test	1	95	120	95	4	0	1	4000000	0	2613873262372	2063217	0	5	7190	2613873262281	0
15404	stress_pi_test	0	98	120	98	1	1	1	68000000	1000000000	2614277663135	804275390	0	488807	1099916308	2614277661502	2614277663135
15405	stress_pi_test	512	97	120	97	2	1	1	60000000	372000000	2614277665336	432562264	0	391038	618248313	2614277664256	2614277666966
15406	stress_pi_test	1	96	120	96	3	1	1	72000000	284000000	2614277668122	617554977	0	488798	224412955	2614277666966	0
15407	stress_pi_test	0	98	120	98	1	2	1	124000000	912000000	2614277670025	806576673	0	477505	1097617765	2614277668320	2614277670025
15408	stress_pi_test	512	97	120	97	2	2	1	72000000	388000000	2614277672475	430590818	0	381998	566127195	2614277671160	2614277674145
15409	stress_pi_test	1	96	120	96	3	2	1	80000000	280000000	2614277675317	617200631	0	477498	177038598	2614277674145	0
15410	stress_pi_test	0	98	120	98	1	3	1	92000000	964000000	2614277678416	806040189	0	471799	1098335261	2614277676335	2614277678416
15411	stress_pi_test	512	97	120	97	2	3	1	40000000	392000000	2614277681053	425434857	0	377437	617971646	2614277679963	0
15412	stress_pi_test	1	96	120	96	3	3	1	68000000	300000000	2614277683747	622981889	0	471797	228274105	2614277682651	0
15413	stress_pi_test	0	98	120	98	1	4	1	132000000	900000000	2614277688585	804075700	0	707917	1100196819	2614277685002	2614277686608
15414	stress_pi_test	0	97	120	97	2	4	1	56000000	328000000	2614277688585	433330469	0	566328	618703511	2614277687649	2614277692005
15415	stress_pi_test	1	96	120	96	3	4	1	80000000	360000000	2614277694517	616950061	0	707911	225779525	2614277692926	0
15416	stress_pi_test	0	98	120	98	1	5	1	104000000	984000000	2614277702319	804497096	0	485752	1099748671	2614277695644	2614277697466
15417	stress_pi_test	0	97	120	97	2	5	1	56000000	344000000	2614277702319	432177832	0	388596	618795209	2614277701123	2614277706045
15418	stress_pi_test	512	96	120	96	3	5	1	48000000	320000000	2614277708585	617731207	0	485746	222009000	2614277706995	2614277711059

Рис. 4.5: Вывод логов в ситуации инверсии приоритетов

Выводы

В результате исследований было выяснено, что процессы проигрывания видеофайлов и аудиофайлов, а также игровые и интерактивные процессы

в ОС Linux планируются в соответствии с алгоритмом SCHED_NORMAL и имеют статический приоритет – 120. Однако в ситуации инверсии приоритетов приоритет низкоприоритетного процесса повышается до приоритета высокоприоритетного процесса.

ЗАКЛЮЧЕНИЕ

В соответствии с заданием на курсовой проект по курсу «Операционные Системы» был разработан загружаемый модуль ядра Linux, предоставляющий информацию о процессах в системе за некоторый промежуток времени: их приоритетах, состояниях, времени выполнения, а также исполняющем ядре процессора. Таким образом, цель была достигнута. Для её достижения были решены следующие задачи:

1. описана работа планировщика Linux;
2. проанализированы и выбраны структуры ядра, содержащие необходимую информацию;
3. проанализированы и выбраны методы передачи информации из модуля ядра в пространство пользователя;
4. разработаны алгоритмы, используемые в программном обеспечении;
5. проведено исследование с помощью разработанного программного обеспечения для выявленения того, планируются ли процессы проигрывания аудио- и видеофайлов, а также игровые и интерактивные процессы в ОС Linux как процессы реального времени.

В результате проведенных исследований с помощью разработанного ПО было показано, что процессы проигрывания аудиофайлов, видеофайлов, а также игровые и интерактивные процессы в ОС Linux не планируются как задачи реального времени. Однако в ситуации инверсии приоритетов приоритет низкоприоритетного процесса повышается до приоритета высокоприоритетного процесса.

Список использованных источников

1. Документация к коду ядра Linux [Электронный ресурс]. — Режим доступа: URL: https://www.kernel.org (дата обращения: 10.11.2022).
2. Документация к Ubuntu [Электронный ресурс]. — Режим доступа: URL: https://manpages.ubuntu.com (дата обращения: 18.11.2022).
3. Исходный код ядра Linux [Электронный ресурс]. — Режим доступа: URL: https://elixir.bootlin.com (дата обращения: 20.11.2022).
4. Электронный сборник технических статей [Электронный ресурс]. — Режим доступа: URL: https://www.programmersought.com (дата обращения: 30.11.2022).
5. Спецификация языка C [Электронный ресурс]. — Режим доступа: URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (дата обращения: 20.11.2022).