

Оглавление

Введение	3
1 Аналитический раздел	4
1.1 Формализация цели	4
1.2 Понятие процесса реального времени	4
1.3 Описание работы RT Scheduler	4
1.4 Структуры ядра	5
1.4.1 task_struct	6
1.4.2 sched_info	11
1.4.3 sched_rt_entity	12
1.5 Способы определения принадлежности задачи к группе задач реального времени	13
1.5.1 Функция rt_prio	13
1.5.2 Функция rt_task	14
1.5.3 Функция task_is_realtime	14
1.6 Передача данных из пространства ядра в пространство пользователя	16
2 Конструкторский раздел	19
2.1 Требования к программному обеспечению	19
2.2 Модуль логирования процессов реального времени	19
2.3 Программа загрузки модуля и получения информации	22
2.4 Схемы работы модуля и дополнительного программного обеспечения	23
3 Технологический раздел	30
3.1 Выбор и обоснование языка программирования и среды разработки	30
3.2 Программа загрузки модуля и получения информации	30
3.3 Модуль логирования процессов реального времени	31
3.4 Makefile	36
3.5 Демонстрация работы	37
4 Исследовательский раздел	42
4.1 Бла-бла	42
ЗАКЛЮЧЕНИЕ	43
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	44
ПРИЛОЖЕНИЕ 1	46

ВВЕДЕНИЕ

Планирование – это управление распределением ресурсов процессора между различными конкурирующими процессами путем передачи им управления согласно некоторой стратегии планирования. За планирование процессов в операционных системах отвечают планировщики задач, каждый из которых может использовать собственный уникальный алгоритм планирования процессов.

Linux – операционная система мягкого реального времени. Классы планирования реального времени, блокировка памяти, разделяемая память и сигналы реального времени получили поддержку в Linux с самых первых дней. Очереди сообщений POSIX, часы и таймеры поддерживаются в ядре версии 2.6. Асинхронный ввод/вывод также поддерживается с самых первых дней, но эта реализация была полностью сделана в библиотеке языка Си пользовательского пространства. Linux версии 2.6 имеет поддержку AIO (Asynchronous I/O) в ядре. Библиотека языка C GNU и glibc также претерпели изменения для поддержки этих расширений реального времени. Для обеспечения лучшей поддержки в Linux POSIX.1b ядро и glibc работают вместе.

Открытый исходный код Linux позволяет самостоятельно изучить особенности системы.

Данная курсовая работа направлена на изучение особенностей планирования процессов реального времени, таких как воспроизведение видео и аудио, и разработку программного обеспечения для мониторинга приоритетов, времени выполнения и простоя процессов на операционной системе Linux. В результате работы будет представлен анализ особенностей планирования при воспроизведении аудио и видеофайлов.

1. Аналитический раздел

В данном разделе приведена формализация цели, понятие процесса реального времени и описание работы планировщика реального времени. Также представлены структуры ядра и пояснения к их полям, которые понадобятся при решении поставленной задачи. Описаны способы определения принадлежности задачи к группе задач реального времени и рассмотрена их реализация.

1.1 Формализация цели

Цель работы – разработать загружаемый модуль ядра для мониторинга приоритетов, времени выполнения и простоя процессов на ОС Linux и проанализировать с использованием данного модуля воспроизведение аудиофайлов и видеофайлов.

Для достижения поставленной цели потребуется:

- 1) проанализировать структуры ядра, позволяющие определить приоритет, время выполнения и простоя процессов;
- 2) проанализировать методы передачи информации из модуля ядра в пространство пользователя;
- 3) спроектировать и реализовать загружаемый модуль ядра;
- 4) проанализировать с использованием реализованного модуля воспроизведение аудиофайлов и видеофайлов.

1.2 Понятие процесса реального времени

Процесс реального времени – это процесс, который имеет приоритет над обычными процессами. В Linux планирование процессов реального времени возложено на так называемый RT Scheduler (Real Time Scheduler – планировщик реального времени), а обычные процессы обрабатываются с использованием CFS Scheduler (Completely Fair Scheduler – совершенно честный планировщик). Работа данных планировщиков строится на работе с группами процессов, которые расположены в очередях, которые, в свою очередь, организованы в дерево. [1]

1.3 Описание работы RT Scheduler

Проблема планирования в режиме реального времени заключается в том, что группы процессов должны полагаться на постоянство объема полосы пропускания (например, времени исполнения). Для планирования несколь-

ких групп задач в реальном времени может потребоваться назначение, так называемого, гарантированного доступного времени исполнения. [2]

В качестве решения поставленной проблемы доступное процессорное время делится посредством указания того, сколько времени на исполнение даётся на указанный период. Данное время выделяется для каждой группы процессов реального времени, причём каждая группа может исполняться только в свое выделенное время. [2]

Время, не выделенное группам реального времени или неиспользованное ими, будет выделено задачами с обычным приоритетом (SCHED_OTHER). [2]

Для примера можно рассмотреть следующую задачу: рендерер реального времени с фиксированным количеством кадров должен выдавать 25 кадров в секунду, что дает период 0.04 секунды на каждый кадр. Если перед рендерером также стоит параллельная задача проигрывания музыки и ответа на ввод, оставляя около 80% доступного процессорного времени, предназначенного для графики, то для этой группы можно выделить время выполнения $0.8 \cdot 0.04 = 0.032$ секунды.

Таким образом, графическая группа будет иметь период 0.04 секунды с ограничением времени выполнения 0.032 секунды. Если аудиопотоку необходимо заполнять буфер DMA (Direct Memory Access, прямой доступ к памяти) каждые 0.005 секунд, но для этого требуется около 3% времени процессора, ему может быть выделено время выполнения $0.03 \cdot 0.005 = 0.00015$ секунд. Таким образом, данная группа может быть запланирована с периодом 0.005 секунд с временем выполнения 0.00015 секунд.

Оставшееся процессорное время будет использовано для ввода данных пользователем и других задач.

Однако на текущий момент приведенный выше пример еще не реализован полностью в силу того, что отсутствует реализация планировщика EDF (Earliest Deadline First scheduling, алгоритм планирования по ближайшему сроку завершения) для использования неоднородных периодов. [2]

1.4 Структуры ядра

Современные операционные системы предоставляют пользователю фундаментальные концепции, такие как, файл или процесс. [3]

С использованием документации представляется возможность полу-

чить доступ к данным концепциям и изучить работу системы изнутри.

1.4.1 task_struct

Процесс состоит из нескольких компонентов [3]:

- стек процесса;
- регистры процессора, в которые загружены ключевые переменные (зависит от архитектуры);
- адресное пространство;
- ресурсы: дескрипторы открытых файлов, ожидающие обработки сигналы;
- управляющие структуры ядра ОС.

Структура в ядре Linux, соответствующая каждому процессу, – `task_struct`. Она определена в файле `include/linux/sched.h`. Все процессы существующие в системе процессы объединены в кольцевой список. [3]

Стоит отметить, что данная структура занимает в памяти порядка 1.7 килобайт.

Поля структуры содержат информацию о процессе, которую можно поделить на несколько категорий [3]:

- поля, отвечающие за общую информацию о процессе (`PID`, `exit_code`, `PPID`);
- поля, востребованные планировщиком задач (`prio`, `static_prio`, `timeslice`);
- поля, связанные с безопасностью (`uid`, `gid`).

Структура `task_struct` для Linux v5.16rc8 представлена в приложении (см. Приложение 1).

Далее будут отмечены наиболее информативные в проводимой работе поля данной структуры и их назначение.

pid (Process Identifier) – уникальный идентификатор процесса. 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 [4].

`prio`, `static_prio`, `normal_prio`, `rt_priority` – приоритеты процесса.

prio – это значение, которое использует планировщик задач при выборе процесса. Чем ниже значение данной переменной, тем выше приоритет про-

цесса (может принимать значения от 0 до 139, то есть MAX_PRIO, значение которого вычисляется с использованием переменной MAX_RT_PRIO со значением 100) [5]. Также данный приоритет может быть поделён на два интервала:

- от 0 до 99 – процесс реального времени;
- от 100 до 139 – обычный процесс.

Также определены функции определения приоритета процесса, которые приведены в листинге 1.

Листинг 1: Функции определения приоритета процесса, определенные в /kernel/sched.c

```
1  #include "sched_idletask.c"
2  #include "sched_fair.c"
3  #include "sched_rt.c"
4  #ifdef CONFIG_SCHED_DEBUG
5  #include "sched_debug.c"
6  #endif
7
8  /*
9   * __normal_prio - return the priority that is based on
10   ↪ the static prio
11   */
12 static inline int __normal_prio(struct task_struct *p) //
13 ↪ _NORMAL_PRIO function, return static priority value
14 {
15     return p->static_prio;
16 }
17
18 /*
19  * Calculate the expected normal priority: i.e. priority
20  * without taking RT-inheritance into account. Might be
21  * boosted by interactivity modifiers. Changes upon fork,
22  * setprio syscalls, and whenever the interactivity
23  * estimator recalculates.
24  */
25 static inline int normal_prio(struct task_struct *p) //
26 ↪ NORMAL_PRIO function
27 {
28     int prio;
```

```

26
27     if (task_has_rt_policy(p)) // The task_has_rt_policy
    ↪ function, the determination process is a real-time
    ↪ process, if the real-time process, returns 1,
    ↪ otherwise returns 0
28         prio = MAX_RT_PRIO-1 - p->rt_priority; // The
    ↪ process is real-time process, and the PRIO
    ↪ value is related to the real-time priority
    ↪ value: PRIO = MAX_RT_PRIO -1 - P-> rt_priority
29     else
30         prio = __normal_prio(p); // The process is a
    ↪ non-real-time process, then the PRIO value is
    ↪ a static priority value, that is, PRIO = P->
    ↪ static_prio
31     return prio;
32 }
33
34 /*
35  * Calculate the current priority, i.e. the priority
36  * taken into account by the scheduler. This value might
37  * be boosted by RT tasks, or might be boosted by
38  * interactivity modifiers. Will be RT if the task got
39  * RT-boosted. If not then it returns p->normal_prio.
40  */
41 static int effective_prio(struct task_struct *p) // The
    ↪ Effective_Prio function, the effective priority of the
    ↪ calculation process, the PRIO value, this value is the
    ↪ priority value used by the final scheduler
42 {
43     p->normal_prio = normal_prio(p); // Calculate the
    ↪ value of Normal_PRIO
44     /*
45      * If we are RT tasks or we were boosted to RT
    ↪ priority,
46      * keep the priority unchanged. Otherwise, update
    ↪ priority
47      * to the normal priority:
48      */
49     if (!rt_prio(p->prio))
50         return p->normal_prio; // If the process is a
    ↪ non-real-time process, return normal_prio
    ↪ value, at this time Normal_Prio = Static_Prio

```

```

51     return p->prio; // Otherwise, the return value is
    ↪ constant, still PRIO value, at this time, PRIO =
    ↪ MAX_RT_PRIO - 1 - P-> RT_Priority
52 }
53
54 /*****
55 void set_user_nice(struct task_struct *p, long nice)
56 {
57     ...
58     p->prio = effective_prio(p); // In the function
    ↪ set_user_nice, call the Effective_Prio function to
    ↪ set the process's PRIO value.
59     ...
60 }

```

Из предоставленного листинга видно, что для процессов реального времени значение приоритета определяется с использованием поля `prio`, а в ином случае – `static_prio`.

static_prio не изменяется ядром при работе планировщика, однако оно может быть изменено с использованием пользовательского приоритета `nice`. Макрос для изменения данного приоритета предоставлен в листинге 2.

Листинг 2: Макрос для изменения `static_prio`.

```

1  /*
2   * Convert user-nice values [ -20 ... 0 ... 19 ]
3   * to static priority [ MAX_RT_PRIO..MAX_RT_PRIO-1 ],
4   * and back.
5   */
6  #define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
7  #define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
8  #define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)
9
10 /*
11  * 'User priority' is the nice value converted to
    ↪ something we
12  * can work with better when scaling various scheduler
    ↪ parameters,

```



```

13     * it's a [ 0 ... 39 ] range.
14     */
15     #define USER_PRIO(p)          ((p)-MAX_RT_PRIO)
16     #define TASK_USER_PRIO(p)    USER_PRIO((p)->static_prio)
17     #define MAX_USER_PRIO        (USER_PRIO(MAX_PRIO))
18
19     / *** /
20     p->static_prio = NICE_TO_PRIO(nice);

```

Таким образом, значение статического приоритета может быть изменено с использованием вызова макроса `NICE_TO_PRIO(nice)`.

normal_prio 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, непосредственно, его `rt_priority`. Функция вычисления нормального приоритета предоставлена в листинге 3.

Листинг 3: Функция вычисления нормального приоритета.

```

1     static inline int normal_prio(struct task_struct *p) //
    ↪     NORMAL_PRIO function
2     {
3         int prio;
4
5         if (task_has_rt_policy(p)) // The task_has_rt_policy
    ↪     function, the determination process is a real-time
    ↪     process, if the real-time process, returns 1,
    ↪     otherwise returns 0
6         prio = MAX_RT_PRIO-1 - p->rt_priority; // The
    ↪     process is real-time process, and the PRIO
    ↪     value is related to the real-time priority
    ↪     value: PRIO = MAX_RT_PRIO -1 - P-> rt_priority
7     else

```

```

8         prio = __normal_prio(p); // The process is a
        ↪ non-real-time process, then the PRIO value is
        ↪ a static priority value, that is, PRIO = P->
        ↪ static_prio
9     return prio;
10 }

```

Важно отметить факт того, что, чем больше значение `rt_priority`, тем выше приоритет процесса.

1.4.2 sched_info

`sched_info` – структура, которая предоставляет информацию о планировании процесса:

- количество запусков процесса на исполнение центральным процессором;
- количество времени, проведенного в ожидании на исполнение;
- время последнего запуска процесса на исполнение центральным процессором;
- время последнего добавления процесса в очередь на исполнение.

Данная структура предоставлена в листинге 5.

Листинг 4: Структура `sched_info`.

```

1     struct sched_info {
2     #ifdef CONFIG_SCHED_INFO
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
19     #endif /* CONFIG_SCHED_INFO */
20 };

```

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.

1.4.3 sched_rt_entity

`sched_rt_entity` – это структура, используемая для группового планирования процессов реального времени, которое включает в себя деревья групп и их очередей. Данное решение позволяет определять выделяемую производительность процессора для определенных групп процессов.

Структура `sched_rt_entity` предоставлена в листинге 5.

Листинг 5: Структура `sched_rt_entity`.

```

1  struct sched_rt_entity {
2      struct list_head         run_list;
3      unsigned long            timeout;
4      unsigned long            watchdog_stamp;
5      unsigned int              time_slice;
6      unsigned short           on_rq;
7      unsigned short           on_list;
8
9      struct sched_rt_entity    *back;
10 #ifdef CONFIG_RT_GROUP_SCHED
11     struct sched_rt_entity    *parent;
12     /* rq on which this entity is (to be) queued: */
13     struct rt_rq               *rt_rq;
14     /* rq "owned" by this entity/group: */
15     struct rt_rq               *my_q;

```

```

16  #endif
17  } __randomize_layout;

```

В данной структуре поле `parent` указывает на процесс более высокого уровня в дереве. `rt_rq` – это очередь, в которой процесс находится, а `my_q` – это очередь дочерних процессов.

Поле `timeout` изменяется так называемым `watchdog` таймером и используется для проверки того, что процесс не занимает процессор дольше, чем это задано в `RLIMIT_RTIME`.

1.5 Способы определения принадлежности задачи к группе задач реального времени

В файле `/include/linux/sched/rt.h` определены функции, позволяющие определить, является ли процесс задачей реального времени.

1.5.1 Функция `rt_prio`

Реализация данной функции приведена в листинге 6.

Листинг 6: Функция `rt_prio`.

```

1  static inline int rt_prio(int prio)
2  {
3      if (unlikely(prio < MAX_RT_PRIO))
4          return 1;
5      return 0;
6  }

```

Представленная функция определяет, является ли процесс задачей реального времени, с использованием значения приоритета. Приоритет задачи сравнивается с максимальным значением приоритета для задачи реального времени. В случае, если условие истинно, процесс относится к задачам реального времени и возвращается единица.

Важно отметить, что макрос `unlikely` в данном случае применяется для оптимизации скорости выполнения сравнения.

1.5.2 Функция `rt_task`

Реализация данной функции приведена в листинге 7.

Листинг 7: Функция `rt_task`.

```
1 static inline int rt_task(struct task_struct *p)
2 {
3     return rt_prio(p->prio);
4 }
```

`rt_task` является оберточной функцией для вызова функции `rt_prio`. Возвращает единицу в том случае, если процесс является задачей реального времени.

1.5.3 Функция `task_is_realtime`

Реализация данной функции приведена в листинге 8.

Листинг 8: Функция `task_is_realtime`.

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

`task_is_realtime` анализирует политику планировщика из преданной структуры `task_struct`. `SCHED_FIFO` и `SCHED_RR` являются, так называемыми, политиками реального времени.

`SCHED_FIFO` – это политика планирования реального времени ”первый вошел, первый вышел”. Данный алгоритм планирования не использует интервалов времени, а процесс выполняется до завершения, если он не забло-

кирован запросом ввода-вывода, вытеснен высокоприоритетным процессом, или он добровольно не отказывается от процессора. [6]

Следует обратить внимание на то, что процесс `SCHED_FIFO`, вытесненный другим процессом с более высоким приоритетом, остается во главе списка с его приоритетом и возобновит выполнение, как только все процессы с более высоким приоритетом будут вновь заблокированы. Также, когда процесс `SCHED_FIFO` готов к выполнению (например, после пробуждения от операции блокировки), он будет вставлен в конец списка с его приоритетом. [6]

Вызов `sched_setscheduler` или `sched_setparam` поставит процесс `SCHED_FIFO` в начало списка. Как следствие, это может вытеснить исполняющийся в данный момент процесс, если его приоритет такой же, как и у исполняющегося процесса. [6]

`SCHED_RR` – это циклическая (Round-Robin) политика планирования реального времени. Она похожа на `SCHED_FIFO` с той лишь разницей, что процессу `SCHED_RR` разрешено работать как максимум время кванта. Если процесс `SCHED_RR` исчерпывает свой квант времени, он помещается в конец списка с его приоритетом. [6]

Процесс `SCHED_RR`, который был вытеснен процессом с более высоким приоритетом, завершит оставшуюся часть своего кванта времени после возобновления выполнения. [6]

Планировщик класса `SCHED_DEADLINE` реализует алгоритм EDF (Earliest Deadline First), который основан на идее выбора для выполнения из очереди ожидающих процессов задачи, наиболее близкой к истечению крайнего расчетного времени. `SCHED_DEADLINE` поддерживает обеспечение работы процессов, требующих выполнения операций в режиме реального времени, предоставляя для подобных задач гарантированное время выполнения, независимо от общего количества обслуживаемых процессов, и реализуя возможность резервирования пропускной способности процессора для процессов. [7]

Как видно из реализации функции, оставшиеся политики планировщика задач не относятся к алгоритмам планирования реального времени.

1.6 Передача данных из пространства ядра в пространство пользователя

Поставленная задача подразумевает под собой передачу информации из пространства ядра в пространство пользователя.

В Linux для передачи данных из пространства ядра в пространство пользователя зачастую используется виртуальная файловая система procfs.

Ключевая идея данной концепции состоит в том, чтобы выделить некоторую часть файловой системы, являющейся общей для всех файловых систем, и поместить ее код на отдельный уровень, из которого вызываются расположенные ниже конкретные файловые системы с целью фактического управления данными. [8]

Все относящиеся к файлам системные вызовы направляются для первичной обработки в адрес виртуальной файловой системы. Эти вызовы, поступающие от пользовательских процессов, являются стандартными POSIX-вызовами, такими как open, read, write, lseek и так далее. Таким образом, VFS обладает “верхним” интерфейсом к пользовательским процессам. [8]

У VFS также существует “нижний” интерфейс к конкретной файловой системе. Этот интерфейс состоит из нескольких десятков вызовов функций, которые VFS способна направлять к каждой файловой системе для достижения конечного результата. Таким образом, чтобы создать новую файловую систему, работающую с VFS, ее разработчики должны предоставить вызовы функций, необходимых VFS. [8]

В Linux procfs предоставляет все ресурсы для реализации интерфейса между пространством пользователя и пространством ядра.

Структура proc_ops определена в файле linux/proc_fs.h и содержит в себе указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством. Поля структуры представлены в листинге 9.

Листинг 9: Структура 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;

```

Функция `copy_to_user`, определенная в файле `linux/uaccess.h`, позволяет копировать блоки данных из пространства ядра в пространство пользователя. Возвращает количество байт, которые не удалось скопировать. [9]

Реализация функции приведена в листинге 10.

Листинг 10: Реализация функции `copy_to_user`.

```

1     static __always_inline unsigned long __must_check
2     copy_to_user(void __user *to, const void *from, unsigned
    ↪   long n)
3     {
4         if (likely(check_copy_size(from, n, true)))
5             n = _copy_to_user(to, from, n);
6         return n;
7     }

```


Вывод

В разделе была формализована задача, а также приведено понятие процесса реального времени. Была описана работа RT Scheduler, или планировщика задач реального времени.

Также приведены структуры ядра, информация о полях которых требуется для решения поставленной задачи: `task_struct`, `sched_info`, `sched_rt_entity`. Определены особенности и различия полей `prio`, `static_prio` и `normal_prio`, что в дальнейшем помогло понять работу функций `rt_prio`, `rt_task` и `task_is_realtime`, которые были отнесены к способам определения принадлежности задач к группе задач реального времени.

Была описана концепция виртуальной файловой системы, приведена структура `proc_ops` и реализация функции `copy_to_user`.

2. Конструкторский раздел

В данном разделе представлены требования к программному обеспечению, сведения о реализуемом модуле и дополнительном программном обеспечении. Также приведены схемы, описывающие работу компонентов.

2.1 Требования к программному обеспечению

Требуется реализовать загружаемый модуль ядра для мониторинга приоритетов, времени выполнения и простоя процессов, который будет получать информацию о процессах в режиме ядра и предоставлять доступ к данной информации через `procfs`.

В целях упрощения работы с данным модулем также потребуется реализовать дополнительную программу, выполняющую загрузку модуля в систему, а также предоставляющую получаемую из `procfs` информацию без дополнительных манипуляций пользователя.

2.2 Модуль логирования процессов реального времени

Для хранения информации о приоритетах, времени выполнения и простоя процессов реального времени выделяется массив символов.

Листинг 11: Массив символов журнала.

```
26  #define LOG_SIZE 262144
27  static char log[LOG_SIZE] = { 0 };
```

Для того, чтобы не допускать переполнений при записи в журнал, каждый раз перед занесением данных в лог проверяется, может ли он в данный момент вместить новые данные.

Листинг 12: Функция проверки переполнения.

```
27  static char log[LOG_SIZE] = { 0 };
28
29  static int checkOverflow(char *fString, char *sString, int
    ↪  maxSize)
30  {
31      int sumLen = strlen(fString) + strlen(sString);
```

```

32
33     if (sumLen >= maxSize)
34     {
35         printk(KERN_ERR "%s not enough space in log (%d
           ↳ needed but %d available)\n", PREFIX, sumLen,
           ↳ maxSize);
36
37         return -ENOMEM;
38     }
39

```

Листинг 13: Пример использования функции проверки переполнения.

```

52         task = &init_task;
53
54         memset(currentString, 0, TEMP_STRING_SIZE);
55         snprintf(currentString, TEMP_STRING_SIZE,
           ↳ "~~~~~: %lu TIME\n",
           ↳ currentPrint);
56
57         checkOverflow(currentString, log, LOG_SIZE);
58

```

Как видно из предоставленного листинга 13 в случае обнаружения переполнения модуль не прекращает свое выполнение.

В листинге 14 приведено заполнение структуры `proc_ops`. Требуется, чтобы функция `yaRead` включала в себя вызов функции `copy_to_user` в целях передачи информации из журнала в пространство пользователя.

Листинг 14: Заполнение структуры `proc_ops`.

```

1     static int yaOpen(struct inode *spInode, struct file
           ↳ *spFile);
2
3     static ssize_t yaRead(struct file *filep, char __user
           ↳ *buf, size_t count, loff_t *offp);

```

```

4
5 static ssize_t yaWrite(struct file *file, const char
   ↪ __user *buf, size_t len, loff_t *offp);
6
7 static int yaRelease(struct inode *spInode, struct file
   ↪ *spFile);
8
9 static struct proc_ops fops = {proc_read : yaRead,
   ↪ proc_write : yaWrite, proc_open : yaOpen, proc_release
   ↪ : yaRelease};

```

В листинге 15 предоставлены функции инициализации и завершения работы модуля.

Листинг 15: Функции инициализации и завершения работы модуля.

```

133 static int __init md_init(void)
134 {
135     if (!(procFile = proc_create(PROC_FS_NAME, 0666, NULL,
   ↪ &fops)))
136     {
137         printk(KERN_ERR "%s proc_create error\n", PREFIX);
138
139         return -EFAULT;
140     }
141
142     kthread = kthread_run(printTasks, NULL,
   ↪ "taskPrintThread");
143     if (IS_ERR(kthread))
144     {
145         printk(KERN_ERR "%s kthread_run error\n", PREFIX);
146
147         return -EFAULT;
148     }
149
150     printk(KERN_INFO "%s module loaded\n", PREFIX);
151
152     return 0;
153 }
154

```

```

155 static void __exit md_exit(void)
156 {
157     remove_proc_entry(PROC_FS_NAME, NULL);
158
159     printk(KERN_INFO "%s Module goes away... It's his
        ↪ final message.\n", PREFIX);
160 }

```

В данном фрагменте можно увидеть, что в функции инициализации модуля требуется запустить отдельный поток для журналирования, так как в ином случае инициализация не будет завершена до окончания выполнения требующегося количества итераций сбора информации.

2.3 Программа загрузки модуля и получения информации

Программа загрузки модуля и вывода полученной информации должна включать в себя два системных вызова: загрузки и выгрузки модуля. Также требуется получить доступ к файлу, создаваемому модулем в `procfs`. Данное действие может быть произведено с использованием вызова `cat`.

В листингах 16–18 предоставлены фрагменты выполнения указанных вызовов.

Листинг 16: Загрузка модуля в систему.

```

22 system("sudo insmod taskInfoGetter.ko");

```

Листинг 17: Выгрузка модуля из системы.

```

45 system("sudo rmmod taskInfoGetter");

```

Листинг 18: Выполнения вызова `cat`.

```

26 filePointer = popen("cat
    ↪ /proc/yakubaProcessAnalyzer", "r");
27

```

```

28         if (filePointer == NULL)
29         {
30             printf("Error: can't execute cat for process
               ↳ analyzer");
31             return ERROR_COMMAND_EXEC;
32         }

```

Функция `open` в листинге 18 открывает процесс, создавая канал, производя `fork` и вызывая командную оболочку. Возвращаемое значение данной функции – это поток ввода-вывода. При этом будет возвращен `NULL`, если вызовы `fork` или `pipe` завершились ошибкой или если невозможно выделить необходимый для этого объем памяти.

2.4 Схемы работы модуля и дополнительного программного обеспечения

На рисунках 2.1–2.5 предоставлены схемы работы модуля и дополнительного программного обеспечения.

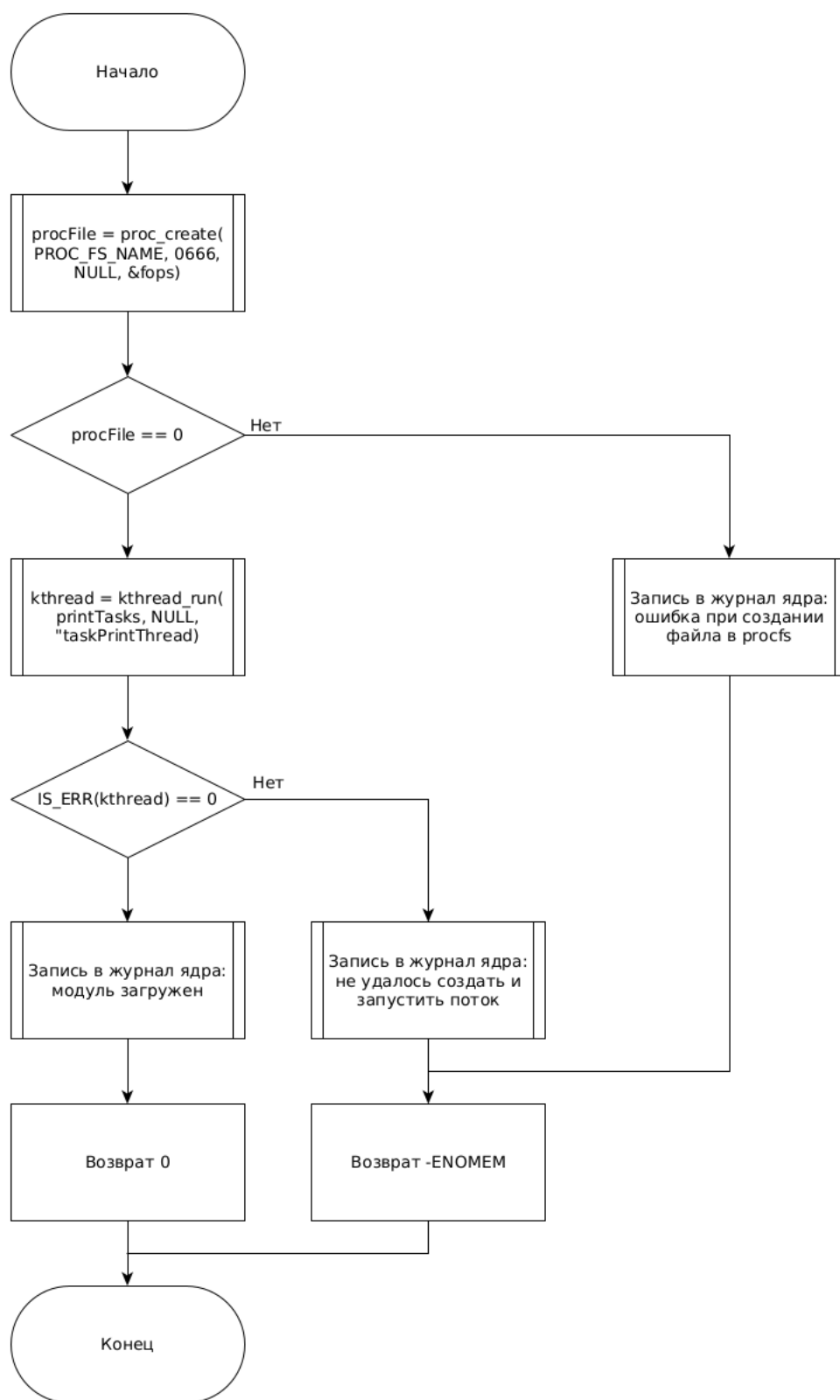


Рис. 2.1: Схема функции инициализации модуля.



Рис. 2.2: Схема функции завершения работы модуля.

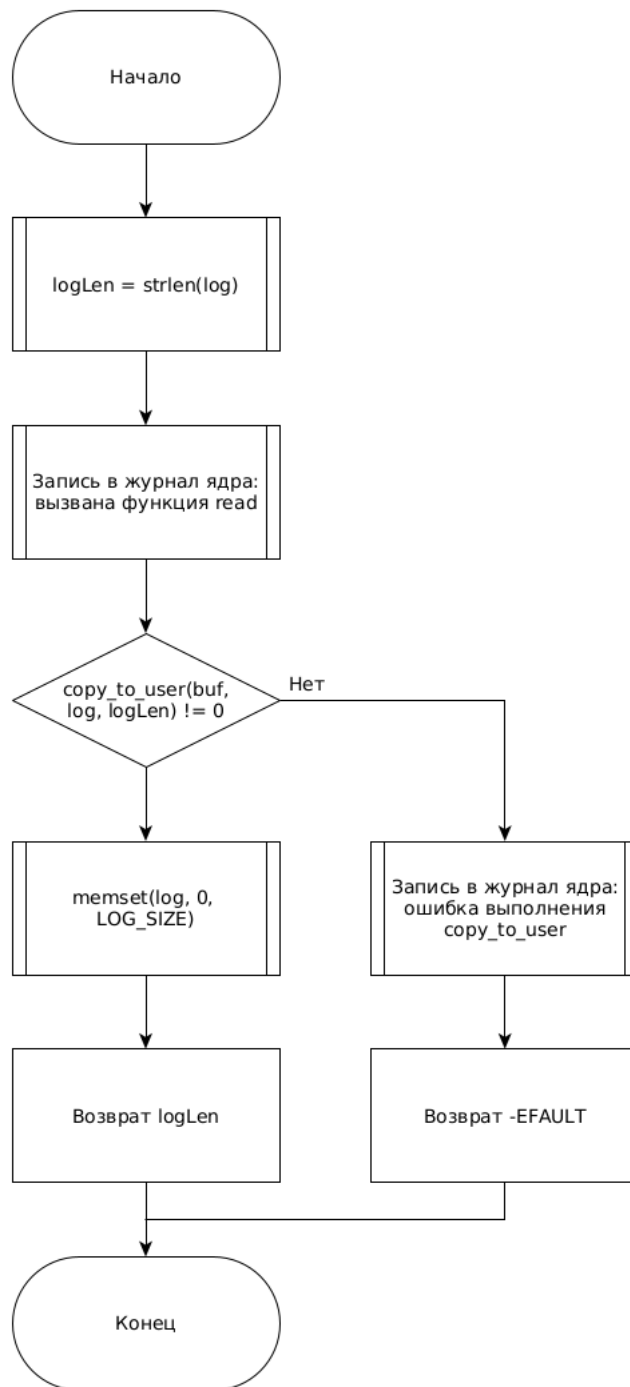


Рис. 2.3: Схема функции `yaRead` обработки чтения из `/proc`.

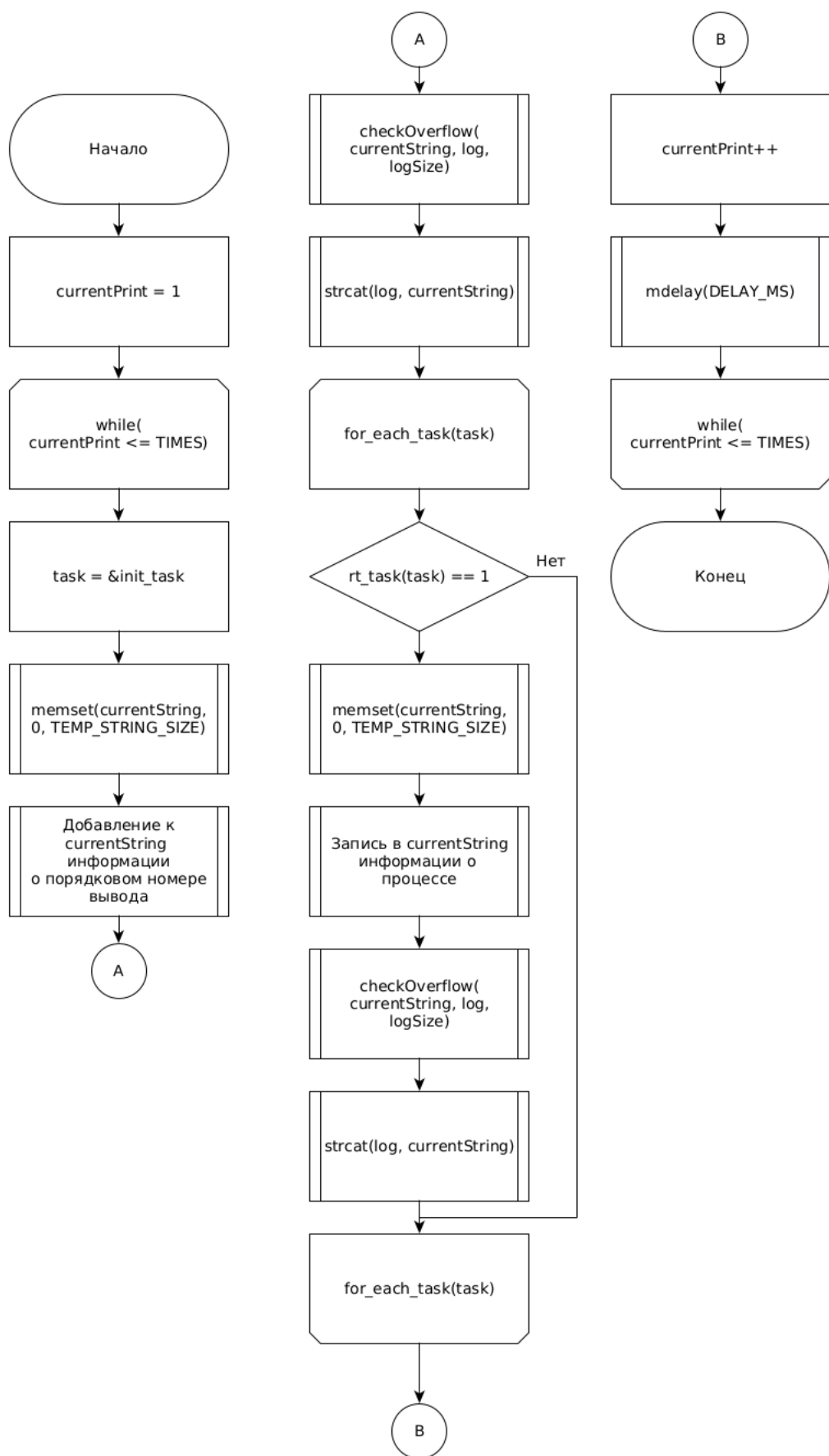


Рис. 2.4: Схема функции printTasks вывода информации о процессах реального времени.

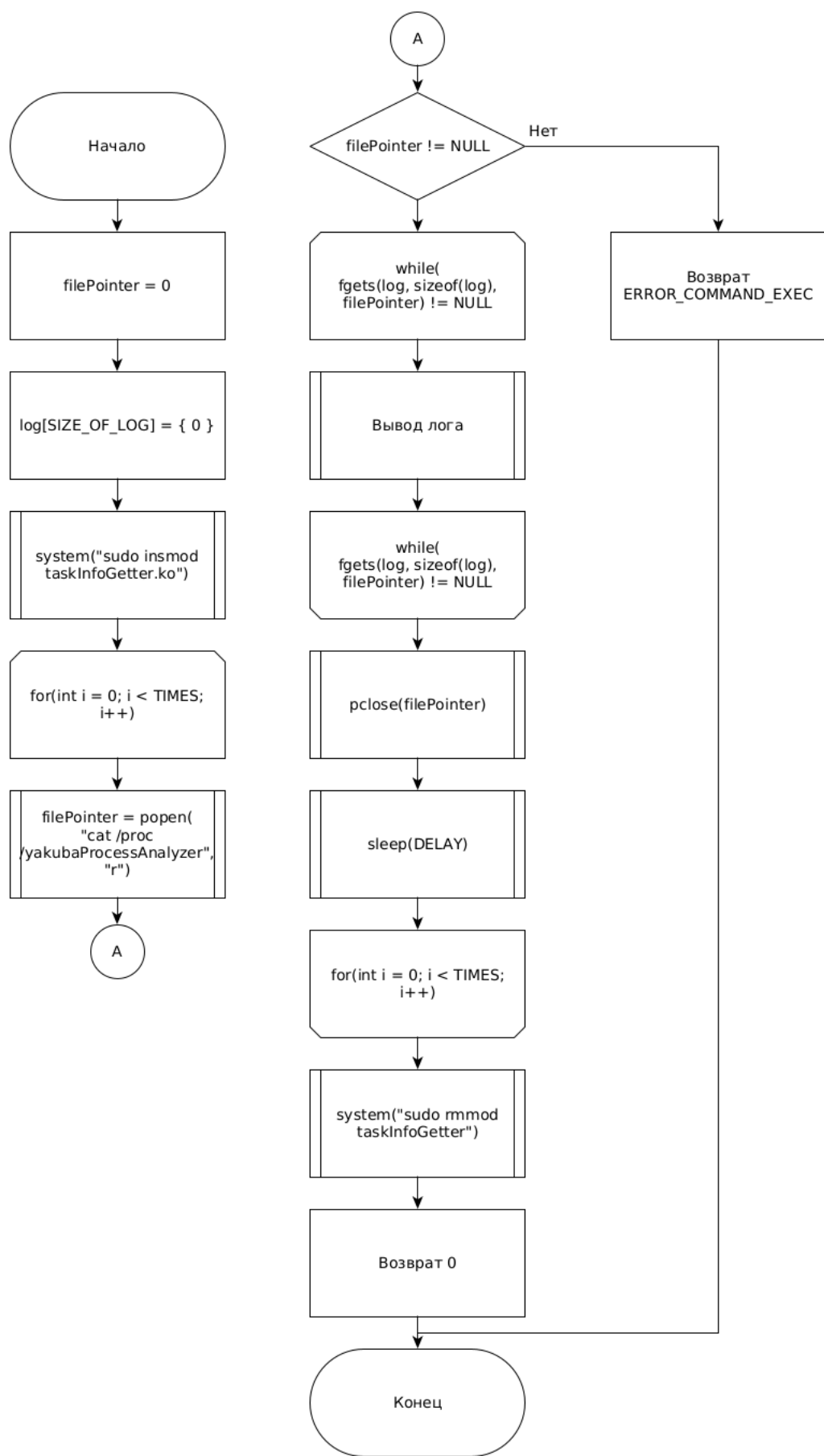


Рис. 2.5: Схема работы программы загрузки модуля и получения информации.

Вывод

В разделе были представлены требования к программному обеспечению. Было обозначено, что требуется реализовать загружаемый модуль ядра для мониторинга приоритетов, времени выполнения и простоя процессов. Также было решено реализовать дополнительную программу загрузки модуля в систему и вывода лога.

Были приведены основные сведения о компонентах, а также схемы, описывающие их работу.

3. Технологический раздел

В данном разделе представлен выбор и обоснование языка программирования и среды разработки, а также листинг реализованных компонентов. Также приведена демонстрация работы.

3.1 Выбор и обоснование языка программирования и среды разработки

При написании программного кода использовался язык программирования C [10].

В качестве среды разработки использовалась “Visual Studio Code” [11]. Данный выбор обусловлен следующими факторами:

- наличие плагинов для написания программ, работающих на уровне ядра,
- широкий функционал,
- ПО с открытым исходным кодом.

3.2 Программа загрузки модуля и получения информации

В листинге 19 предоставлена реализация программы загрузки модуля и получения информации.

Листинг 19: Программа загрузки модуля и получения информации.

```
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
8  #include <errno.h>
9
10 #define SIZE_OF_LOG 8192
11
12 #define ERROR_COMMAND_EXEC 1
13
14 #define TIMES 6
15 #define DELAY 10
16
17 int main(int argc, char *argv[])
```

```

18  {
19      FILE *filePointer = NULL;
20      char log[SIZE_OF_LOG] = {'\0'};
21
22      system("sudo insmod taskInfoGetter.ko");
23
24      for (int i = 0; i < TIMES; i++)
25      {
26          filePointer = popen("cat
27              ↪ /proc/yakubaProcessAnalyzer", "r");
28
29          if (filePointer == NULL)
30          {
31              printf("Error: can't execute cat for process
32              ↪ analyzer");
33              return ERROR_COMMAND_EXEC;
34          }
35
36          while (fgets(log, sizeof(log), filePointer) !=
37              ↪ NULL)
38          {
39              printf("%s", log);
40              fgets(log, sizeof(log), filePointer);
41              printf("%s", log);
42          }
43
44          pclose(filePointer);
45          sleep(DELAY);
46      }
47
48      system("sudo rmmod taskInfoGetter");
49
50      return EXIT_SUCCESS;
51  }

```

3.3 Модуль логирования процессов реального времени

В листинге 20 предоставлена реализация модуля логирования процессов реального времени.

Листинг 20: Модуль логирования процессов реального времени.

```
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
8  #include <linux/kthread.h>
9
10 #define PROC_FS_NAME "yakubaProcessAnalyzer"
11
12 #define PREFIX "~~[TASK INFO]~~:"
13
14 #define TEMP_STRING_SIZE 512
15
16 MODULE_LICENSE("GPL");
17 MODULE_AUTHOR("Yakuba D.");
18
19 #define TIMES 6
20 #define DELAY_MS 10 * 1000
21
22 static struct proc_dir_entry *procFile;
23
24 static struct task_struct *kthread;
25
26 #define LOG_SIZE 262144
27 static char log[LOG_SIZE] = { 0 };
28
29 static int checkOverflow(char *fString, char *sString, int
    ↪ maxSize)
30 {
31     int sumLen = strlen(fString) + strlen(sString);
32
33     if (sumLen >= maxSize)
34     {
35         printk(KERN_ERR "%s not enough space in log (%d
    ↪ needed but %d available)\n", PREFIX, sumLen,
    ↪ maxSize);
```

```

36
37         return -ENOMEM;
38     }
39
40     return 0;
41 }
42
43 static int printTasks(void *arg)
44 {
45     struct task_struct *task;
46     size_t currentPrint = 1;
47
48     char currentString[TEMP_STRING_SIZE];
49
50     while (currentPrint <= TIMES)
51     {
52         task = &init_task;
53
54         memset(currentString, 0, TEMP_STRING_SIZE);
55         snprintf(currentString, TEMP_STRING_SIZE,
56             ↪ "~~~~~: %lu TIME\n",
57             ↪ currentPrint);
58
59         checkOverflow(currentString, log, LOG_SIZE);
60
61         strcat(log, currentString);
62
63         for_each_process(task)
64         {
65             if (rt_task(task))
66             {
67                 memset(currentString, 0,
68                     ↪ TEMP_STRING_SIZE);
69                 snprintf(currentString, TEMP_STRING_SIZE,
70                     "procID: %-5d, name: %15s\nprio:
71                     ↪ %3d, static_prio: %3d,
72                     ↪ normal_prio (with "
73                     "scheduler policy): %3d,
74                     ↪ realtime_prio: %3d\n"
75                     "delay: %10lld\nutime: %10lld
76                     ↪ (ticks), stime: %15lld
77                     ↪ (ticks)\n"

```



```

70         "Sched_rt_entity: timeout: %ld,
        ↪ watchdog_stamp: %ld,
        ↪ time_slice: %d\n\n",
71 task->pid, task->comm,
        ↪ task->prio,
        ↪ task->static_prio,
        ↪ task->normal_prio,
        ↪ task->rt_priority,
72 task->sched_info.run_delay,
        ↪ task->utime, task->stime,
        ↪ task->rt.timeout,
73 task->rt.watchdog_stamp,
        ↪ task->rt.time_slice);
74
75         checkOverflow(currentString, log,
        ↪ LOG_SIZE);
76
77         strcat(log, currentString);
78     }
79 }
80
81     currentPrint++;
82     mdelay(DELAY_MS);
83 }
84
85     return 0;
86 }
87
88 static int yaOpen(struct inode *spInode, struct file
    ↪ *spFile)
89 {
90     printk(KERN_INFO "%s open called\n", PREFIX);
91
92     try_module_get(THIS_MODULE);
93
94     return 0;
95 }
96
97 static ssize_t yaRead(struct file *filep, char __user
    ↪ *buf, size_t count, loff_t *offp)
98 {

```

```

99         ssize_t logLen = strlen(log);
100
101         printk(KERN_INFO "%s read called\n", PREFIX);
102
103         if (copy_to_user(buf, log, logLen))
104         {
105             printk(KERN_ERR "%s copy_to_user error\n",
106                 ↪ PREFIX);
107
108             return -EFAULT;
109         }
110
111         memset(log, 0, LOG_SIZE);
112
113         return logLen;
114     }
115
116     static ssize_t yaWrite(struct file *file, const char
117     ↪ __user *buf, size_t len, loff_t *offp)
118     {
119         printk(KERN_INFO "%s write called\n", PREFIX);
120
121         return 0;
122     }
123
124     static int yaRelease(struct inode *spInode, struct file
125     ↪ *spFile)
126     {
127         printk(KERN_INFO "%s release called\n", PREFIX);
128
129         module_put(THIS_MODULE);
130
131         return 0;
132     }
133
134     static struct proc_ops fops = {proc_read : yaRead,
135     ↪ proc_write : yaWrite, proc_open : yaOpen, proc_release
136     ↪ : yaRelease};
137
138     static int __init md_init(void)
139     {

```

```

135     if (!(procFile = proc_create(PROC_FS_NAME, 0666, NULL,
    ↪      &fops)))
136     {
137         printk(KERN_ERR "%s proc_create error\n", PREFIX);
138
139         return -EFAULT;
140     }
141
142     kthread = kthread_run(printTasks, NULL,
    ↪      "taskPrintThread");
143     if (IS_ERR(kthread))
144     {
145         printk(KERN_ERR "%s kthread_run error\n", PREFIX);
146
147         return -EFAULT;
148     }
149
150     printk(KERN_INFO "%s module loaded\n", PREFIX);
151
152     return 0;
153 }
154
155 static void __exit md_exit(void)
156 {
157     remove_proc_entry(PROC_FS_NAME, NULL);
158
159     printk(KERN_INFO "%s Module goes away... It's his
    ↪      final message.\n", PREFIX);
160 }
161
162 module_init(md_init);
163 module_exit(md_exit);

```

3.4 Makefile

В листинге 21 предоставлено содержание Makefile для сборки компонентов.

Листинг 21: Содержание Makefile.

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := taskInfoGetter.o
3      taskInfoGetter-objs := md.o
4  else
5      CURRENT = $(shell uname -r)
6      KDIR = /lib/modules/$(CURRENT)/build
7      PWD = $(shell pwd)
8
9  default:
10     $(MAKE) -C $(KDIR) M=$(PWD) modules
11     make clean
12
13  logger:
14     gcc starterLogger.c -Wall -Werror -o sl.exe
15
16  clean:
17     @rm -f *.o *.cmd *.flags *.mod.c *.order *.mod
18     ↪ *.symvers
19     @rm -f *.*.cmd *~ *.*~ TODO.*
20     @rm -fR .tmp*
21     @rm -rf .tmp_versions
22
23  disclean: clean
24     @rm *.ko *.symvers *.mod
25
26  endif
27
28  # sudo insmod md.ko
29  # lsmod | grep md
30  # sudo dmesg
31  # sudo rmmod md.ko
```

3.5 Демонстрация работы

На рисунках 3.1–3.3 предоставлена демонстрация работы программы загрузки модуля и получения информации.

На рисунке 3.4 предоставлено содержание журнала ядра при загрузке и выгрузке модуля из системы.


```
procID: 13 , name: rcu_preempt
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 49596120
utime: 0 (ticks), stime: 333333 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 14 , name: rcub/0
prio: 96, static_prio: 120, normal_prio (with scheduler policy): 96, realtime_prio: 1
delay: 677
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 15 , name: rcuc/0
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 14609
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 16 , name: migration/0
prio: 0, static_prio: 120, normal_prio (with scheduler policy): 0, realtime_prio: 99
delay: 6583
utime: 0 (ticks), stime: 1866525 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 17 , name: idle_inject/0
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 288287
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 21 , name: idle_inject/1
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 1022
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 22 , name: migration/1
prio: 0, static_prio: 120, normal_prio (with scheduler policy): 0, realtime_prio: 99
delay: 12456
utime: 0 (ticks), stime: 3331658 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 23 , name: rcuc/1
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 931
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 28 , name: idle_inject/2
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 7335
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30
```

Рис. 3.2: Вывод при запуске starterLogger.c (вторая итерация).

```
.....: 3 TIME
procID: 13 , name: rcu_preempt
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 49727587
utime: 0 (ticks), stime: 333333 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 14 , name: rcub/0
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 677
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 15 , name: rcuc/0
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 14609
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 16 , name: migration/0
prio: 0, static_prio: 120, normal_prio (with scheduler policy): 0, realtime_prio: 99
delay: 6583
utime: 0 (ticks), stime: 1866525 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 17 , name: idle_inject/0
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 288287
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 21 , name: idle_inject/1
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 1022
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 22 , name: migration/1
prio: 0, static_prio: 120, normal_prio (with scheduler policy): 0, realtime_prio: 99
delay: 12456
utime: 0 (ticks), stime: 3331058 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 23 , name: rcuc/1
prio: 98, static_prio: 120, normal_prio (with scheduler policy): 98, realtime_prio: 1
delay: 931
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30

procID: 28 , name: idle_inject/2
prio: 49, static_prio: 120, normal_prio (with scheduler policy): 49, realtime_prio: 50
delay: 7355
utime: 0 (ticks), stime: 0 (ticks)
Sched_rt_entity: timeout: 0, watchdog_stamp: 0, time_slice: 30
```

Рис. 3.3: Вывод при запуске starterLogger.c (третья итерация).

```

[ 334.967144] ~~[TASK INFO]~~: module loaded
[ 334.967506] audit: type=1106 audit(1643887178.936:107): pid=3143 uid=1000 auid=1000 ses:
rminal=/dev/pts/1 res=success'
[ 334.967573] audit: type=1104 audit(1643887178.936:108): pid=3143 uid=1000 auid=1000 ses:
rminal=/dev/pts/1 res=success'
[ 334.972749] ~~[TASK INFO]~~: open called
[ 334.972770] ~~[TASK INFO]~~: read called
[ 334.972797] ~~[TASK INFO]~~: read called
[ 334.972815] ~~[TASK INFO]~~: release called
[ 344.975590] ~~[TASK INFO]~~: open called
[ 344.975602] ~~[TASK INFO]~~: read called
[ 344.975616] ~~[TASK INFO]~~: read called
[ 344.975625] ~~[TASK INFO]~~: release called
[ 346.146147] audit: type=1131 audit(1643887190.116:109): pid=1 uid=0 auid=4294967295 ses:
ccess'
[ 346.178521] audit: type=1334 audit(1643887190.149:110): prog-id=26 op=UNLOAD
[ 346.178523] audit: type=1334 audit(1643887190.149:111): prog-id=25 op=UNLOAD
[ 346.178524] audit: type=1334 audit(1643887190.149:112): prog-id=24 op=UNLOAD
[ 354.978103] ~~[TASK INFO]~~: open called
[ 354.978114] ~~[TASK INFO]~~: read called
[ 354.978128] ~~[TASK INFO]~~: read called
[ 354.978136] ~~[TASK INFO]~~: release called
[ 364.997557] audit: type=1101 audit(1643887208.966:113): pid=3192 uid=1000 auid=1000 ses:
terminal=/dev/pts/1 res=success'
[ 364.998537] audit: type=1110 audit(1643887208.969:114): pid=3192 uid=1000 auid=1000 ses:
ddr=? terminal=/dev/pts/1 res=success'
[ 365.002335] audit: type=1105 audit(1643887208.972:115): pid=3192 uid=1000 auid=1000 ses:
minal=/dev/pts/1 res=success'
[ 365.003711] ~~[TASK INFO]~~: Module goes away... It's his final message.
[ 365.022717] audit: type=1106 audit(1643887208.992:116): pid=3192 uid=1000 auid=1000 ses:

```

Рис. 3.4: Содержание журнала ядра.

Вывод

В качестве средств реализации был выбран язык программирования С. Используемая среда разработки – “Visual Studio Code”.

Были реализованы модуль логирования процессов реального времени и программа загрузки модуля и получения информации, представлены их листинги. Была проведена демонстрация работы реализованного программного обеспечения.

4. Исследовательский раздел

В данном разделе...

4.1 Бла-бла

Мы такие умные, куча исследований...

Вывод

В разделе...

ЗАКЛЮЧЕНИЕ

Во время выполнения курсового проекта были достигнуты поставленные задачи:

- хоп,
- хоп,
- хоп,
- хоп,
- хоп.

Проведённая аналитическая работа позволила...

В результате работы, проведенной в конструкторском разделе, были приведены... Также была определена схема работы...

Для реализации в качестве используемого языка программирования был выбран ЯП ..., а в качестве среды разработки – ...

В результате работы было...

В ходе выполнения поставленных задач были получены знания в области ..., а также изучены...

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Список литературы

1. Fatal Errors: Linux Process Scheduling-Real-Time Scheduler Learning Notes [Электронный ресурс]. Режим доступа: https://www.fatalerrors.org/a/linux-process-scheduling-real-time-scheduler_learning-notes.html (дата обращения 20.12.2021).
2. The Linux Kernel documentation [Электронный ресурс]. Режим доступа: <https://www.kernel.org> (дата обращения 20.12.2021).
3. А. Кирьянов Д. Модель процессов в современных операционных системах и их реализация в ОС Linux // Сервис в России и за рубежом. 2007. № 1.
4. Проект Losst [Электронный ресурс]. Режим доступа: <https://losst.ru> (дата обращения 20.12.2021).
5. Programmer All: Linux kernel learning notes (6) [Электронный ресурс]. Режим доступа: <https://www.programmerall.com/article/63151049863/> (дата обращения 20.12.2021).
6. Разработка и внедрение системы на встраиваемом Linux: Планирование процессов [Электронный ресурс]. Режим доступа: <http://dmilvdv.narod.ru/Translate/ELSDD/index.html> (дата обращения 20.12.2021).
7. Проект OpenNET [Электронный ресурс]. Режим доступа: <https://www.opennet.ru/opennews/art.shtml?num=38906> (дата обращения 20.12.2021).
8. Таненбаум Х. Бос Х. Современные операционные системы. СПб.: Питер, 2015. с. 1120.

9. Лекции университета Бернингема [Электронный ресурс]. Режим доступа: <https://www.birmingham.ac.uk/schools/computer-science/index.aspx> (дата обращения 22.12.2021).
10. Спецификация языка C [Электронный ресурс]. Режим доступа: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf> (дата обращения 22.12.2021).
11. Документация Visual Studio Code [Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/docs> (дата обращения 22.12.2021).

ПРИЛОЖЕНИЕ 1

Листинг 22: Структура ядра task_struct

```
1  struct task_struct {
2  #ifdef CONFIG_THREAD_INFO_IN_TASK
3      /*
4       * For reasons of header soup (see
5       * ↪ current_thread_info()), this
6       * must be the first element of task_struct.
7       */
8      struct thread_info      thread_info;
9  #endif
10     unsigned int             __state;
11
12     #ifdef CONFIG_PREEMPT_RT
13         /* saved state for "spinlock sleepers" */
14         unsigned int         saved_state;
15     #endif
16
17     /*
18      * This begins the randomizable portion of
19      * ↪ task_struct. Only
20      * scheduling-critical items should be added above
21      * ↪ here.
22      */
23     randomized_struct_fields_start
24
25     void                    *stack;
26     refcount_t              usage;
27     /* Per task flags (PF_*), defined further below: */
28     unsigned int            flags;
29     unsigned int            ptrace;
30
31     #ifdef CONFIG_SMP
32         int                 on_cpu;
33         struct __call_single_node wake_entry;
34         unsigned int         wakee_flips;
35         unsigned long         wakee_flip_decay_ts;
36         struct task_struct    *last_wakee;
```

```

34
35     /*
36     * recent_used_cpu is initially set as the last CPU
37     *   ↳ used by a task
38     * that wakes affine another task. Waker/wakee
39     *   ↳ relationships can
40     * push tasks around a CPU where each wakeup moves to
41     *   ↳ the next one.
42     * Tracking a recently used CPU allows a quick search
43     *   ↳ for a recently
44     * used CPU that may be idle.
45     */
46
47     int                recent_used_cpu;
48     int                wake_cpu;
49 #endif
50
51     int                on_rq;
52
53     int                prio;
54     int                static_prio;
55     int                normal_prio;
56     unsigned int       rt_priority;
57
58     struct sched_entity se;
59     struct sched_rt_entity rt;
60     struct sched_dl_entity dl;
61     const struct sched_class *sched_class;
62
63 #ifdef CONFIG_SCHED_CORE
64     struct rb_node      core_node;
65     unsigned long       core_cookie;
66     unsigned int       core_occupation;
67 #endif
68
69 #ifdef CONFIG_CGROUP_SCHED
70     struct task_group    *sched_task_group;
71 #endif
72
73 #ifdef CONFIG_UCLAMP_TASK
74     /*
75     * Clamp values requested for a scheduling entity.
76     * Must be updated with task_rq_lock() held.
77     */
78

```

```

71     */
72     struct uclamp_se          uclamp_req[UCLAMP_CNT];
73     /*
74     * Effective clamp values used for a scheduling
75     ↪ entity.
76     * Must be updated with task_rq_lock() held.
77     */
78     struct uclamp_se          uclamp[UCLAMP_CNT];
79 #endif
80
81     struct sched_statistics    stats;
82
83 #ifdef CONFIG_PREEMPT_NOTIFIERS
84     /* List of struct preempt_notifier: */
85     struct hlist_head          preempt_notifiers;
86 #endif
87
88 #ifdef CONFIG_BLK_DEV_IO_TRACE
89     unsigned int               btrace_seq;
90 #endif
91
92     unsigned int               policy;
93     int                        nr_cpus_allowed;
94     const cpumask_t            *cpus_ptr;
95     cpumask_t                  *user_cpus_ptr;
96     cpumask_t                  cpus_mask;
97     void                       *migration_pending;
98 #ifdef CONFIG_SMP
99     unsigned short             migration_disabled;
100 #endif
101
102     unsigned short             migration_flags;
103
104 #ifdef CONFIG_PREEMPT_RCU
105     int                        rcu_read_lock_nesting;
106     union rcu_special           rcu_read_unlock_special;
107     struct list_head           rcu_node_entry;
108     struct rcu_node             *rcu_blocked_node;
109 #endif /* #ifdef CONFIG_PREEMPT_RCU */
110
111 #ifdef CONFIG_TASKS_RCU
112     unsigned long              rcu_tasks_nvcsw;

```

```

111         u8                rcu_tasks_holdout;
112         u8                rcu_tasks_idx;
113         int               rcu_tasks_idle_cpu;
114         struct list_head   rcu_tasks_holdout_list;
115     #endif /* #ifdef CONFIG_TASKS_RCU */
116
117     #ifdef CONFIG_TASKS_TRACE_RCU
118         int               trc_reader_nesting;
119         int               trc_ipi_to_cpu;
120         union rcu_special   trc_reader_special;
121         bool              trc_reader_checked;
122         struct list_head   trc_holdout_list;
123     #endif /* #ifdef CONFIG_TASKS_TRACE_RCU */
124
125         struct sched_info   sched_info;
126
127         struct list_head   tasks;
128     #ifdef CONFIG_SMP
129         struct plist_node   pushable_tasks;
130         struct rb_node      pushable_dl_tasks;
131     #endif
132
133         struct mm_struct    *mm;
134         struct mm_struct    *active_mm;
135
136         /* Per-thread vma caching: */
137         struct vmacache     vmacache;
138
139     #ifdef SPLIT_RSS_COUNTING
140         struct task_rss_stat   rss_stat;
141     #endif
142         int                exit_state;
143         int                exit_code;
144         int                exit_signal;
145         /* The signal sent when the parent dies: */
146         int                pdeath_signal;
147         /* JOBCTL_*, siglock protected: */
148         unsigned long       jobctl;
149
150         /* Used for emulating ABI behavior of previous Linux
        ↪ versions: */

```



```

151         unsigned int                personality;
152
153         /* Scheduler bits, serialized by scheduler locks: */
154         unsigned                    sched_reset_on_fork:1;
155         unsigned                    sched_contributes_to_load:1;
156         unsigned                    sched_migrated:1;
157     #ifdef CONFIG_PSI
158         unsigned                    sched_psi_wake_requeue:1;
159     #endif
160
161         /* Force alignment to the next boundary: */
162         unsigned                    :0;
163
164         /* Unserialized, strictly 'current' */
165
166         /*
167          * This field must not be in the scheduler word above
168          *   ↳ due to wakelist
169          *   ↳ queueing no longer being serialized by p->on_cpu.
170          *   ↳ However:
171          *
172          *   p->XXX = X;                ttwu()
173          *   schedule()                 if (p->on_rq && ..) //
174          *   ↳ false
175          *   smp_mb__after_spinlock();    if
176          *   ↳ (smp_load_acquire(&p->on_cpu) && //true
177          *   deactivate_task()
178          *   ↳ ttwu_queue_wakelist())
179          *   p->on_rq = 0;                p->sched_remote_wakeup
180          *   ↳ = Y;
181          *
182          * guarantees all stores of 'current' are visible
183          *   ↳ before
184          *   ->sched_remote_wakeup gets used, so it can be in
185          *   ↳ this word.
186          */
187         unsigned                    sched_remote_wakeup:1;
188
189         /* Bit to tell LSMs we're in execve(): */
190         unsigned                    in_execve:1;
191         unsigned                    in_iowait:1;

```

```

184 #ifndef TIF_RESTORE_SIGMASK
185     unsigned                restore_sigmask:1;
186 #endif
187 #ifdef CONFIG_MEMCG
188     unsigned                in_user_fault:1;
189 #endif
190 #ifdef CONFIG_COMPAT_BRK
191     unsigned                brk_randomized:1;
192 #endif
193 #ifdef CONFIG_CGROUPS
194     /* disallow userland-initiated cgroup migration */
195     unsigned                no_cgroup_migration:1;
196     /* task is frozen/stopped (used by the cgroup freezer)
       ↪ */
197     unsigned                frozen:1;
198 #endif
199 #ifdef CONFIG_BLK_CGROUP
200     unsigned                use_memdelay:1;
201 #endif
202 #ifdef CONFIG_PSI
203     /* Stalled due to lack of memory */
204     unsigned                in_memstall:1;
205 #endif
206 #ifdef CONFIG_PAGE_OWNER
207     /* Used by page_owner=on to detect recursion in page
       ↪ tracking. */
208     unsigned                in_page_owner:1;
209 #endif
210 #ifdef CONFIG_EVENTFD
211     /* Recursion prevention for eventfd_signal() */
212     unsigned                in_eventfd_signal:1;
213 #endif
214
215     unsigned long           atomic_flags; /* Flags
       ↪ requiring atomic access. */
216
217     struct restart_block    restart_block;
218
219     pid_t                   pid;
220     pid_t                   tgid;
221

```

```

222 #ifdef CONFIG_STACKPROTECTOR
223     /* Canary value for the -fstack-protector GCC feature:
224     ↪ */
225     unsigned long          stack_canary;
226 #endif
227     /*
228     * Pointers to the (original) parent process, youngest
229     ↪ child, younger sibling,
230     * older sibling, respectively. (p->father can be
231     ↪ replaced with
232     * p->real_parent->pid)
233     */
234
235     /* Real parent process: */
236     struct task_struct __rcu    *real_parent;
237
238     /* Recipient of SIGCHLD, wait4() reports: */
239     struct task_struct __rcu    *parent;
240
241     /*
242     * Children/sibling form the list of natural children:
243     */
244     struct list_head          children;
245     struct list_head          sibling;
246     struct task_struct        *group_leader;
247
248     /*
249     * 'ptraced' is the list of tasks this task is using
250     ↪ ptrace() on.
251     *
252     * This includes both natural children and
253     ↪ PTRACE_ATTACH targets.
254     * 'ptrace_entry' is this task's link on the
255     ↪ p->parent->ptraced list.
256     */
257     struct list_head          ptraced;
258     struct list_head          ptrace_entry;
259
260     /* PID/PID hash table linkage. */
261     struct pid                *thread_pid;
262     struct hlist_node          pid_links[PIDTYPE_MAX];

```

```

257     struct list_head      thread_group;
258     struct list_head      thread_node;
259
260     struct completion      *vfork_done;
261
262     /* CLONE_CHILD_SETTID: */
263     int __user              *set_child_tid;
264
265     /* CLONE_CHILD_CLEARTID: */
266     int __user              *clear_child_tid;
267
268     /* PF_IO_WORKER */
269     void                    *pf_io_worker;
270
271     u64                     utime;
272     u64                     stime;
273 #ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
274     u64                     utimescaled;
275     u64                     stimescaled;
276 #endif
277     u64                     gtime;
278     struct prev_cputime      prev_cputime;
279 #ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
280     struct vtime             vtime;
281 #endif
282
283 #ifdef CONFIG_NO_HZ_FULL
284     atomic_t                 tick_dep_mask;
285 #endif
286     /* Context switch counts: */
287     unsigned long            nvcs;
288     unsigned long            nivcs;
289
290     /* Monotonic time in nsecs: */
291     u64                      start_time;
292
293     /* Boot based time in nsecs: */
294     u64                      start_boottime;
295
296     /* MM fault and swap info: this can arguably be seen
    ↪ as either mm-specific or thread-specific: */

```

```

297         unsigned long             min_flt;
298         unsigned long             maj_flt;
299
300         /* Empty if CONFIG_POSIX_CPUTIMERS=n */
301         struct posix_cputimers     posix_cputimers;
302
303     #ifdef CONFIG_POSIX_CPU_TIMERS_TASK_WORK
304         struct posix_cputimers_work  posix_cputimers_work;
305     #endif
306
307         /* Process credentials: */
308
309         /* Tracer's credentials at attach: */
310         const struct cred __rcu      *ptracer_cred;
311
312         /* Objective and real subjective task credentials
313          * ↪ (COW): */
314         const struct cred __rcu      *real_cred;
315
316         /* Effective (overridable) subjective task credentials
317          * ↪ (COW): */
318         const struct cred __rcu      *cred;
319
320     #ifdef CONFIG_KEYS
321         /* Cached requested key. */
322         struct key                   *cached_requested_key;
323     #endif
324
325         /*
326          * executable name, excluding path.
327          *
328          * - normally initialized setup_new_exec()
329          * - access it with [gs]et_task_comm()
330          * - lock it with task_lock()
331          */
332         char                         comm[TASK_COMM_LEN];
333
334         struct nameidata             *nameidata;
335
336     #ifdef CONFIG_SYSVIPC
337         struct sysv_sem              sysvsem;

```

```

336     struct sysv_shm                sysvshm;
337 #endif
338 #ifdef CONFIG_DETECT_HUNG_TASK
339     unsigned long                  last_switch_count;
340     unsigned long                  last_switch_time;
341 #endif
342     /* Filesystem information: */
343     struct fs_struct                *fs;
344
345     /* Open file information: */
346     struct files_struct             *files;
347
348 #ifdef CONFIG_IO_URING
349     struct io_uring_task            *io_uring;
350 #endif
351
352     /* Namespaces: */
353     struct nsproxy                  *nsproxy;
354
355     /* Signal handlers: */
356     struct signal_struct             *signal;
357     struct sighand_struct __rcu      *sighand;
358     sigset_t                        blocked;
359     sigset_t                        real_blocked;
360     /* Restored if set_restore_sigmask() was used: */
361     sigset_t                        saved_sigmask;
362     struct sigpending                pending;
363     unsigned long                   sas_ss_sp;
364     size_t                          sas_ss_size;
365     unsigned int                    sas_ss_flags;
366
367     struct callback_head             *task_works;
368
369 #ifdef CONFIG_AUDIT
370 #ifdef CONFIG_AUDITSYSCALL
371     struct audit_context             *audit_context;
372 #endif
373     kuid_t                          loginuid;
374     unsigned int                    sessionid;
375 #endif
376     struct seccomp                   seccomp;

```

```

377     struct syscall_user_dispatch    syscall_dispatch;
378
379     /* Thread group tracking: */
380     u64                parent_exec_id;
381     u64                self_exec_id;
382
383     /* Protection against (de-)allocation: mm, files, fs,
384     ↪  tty, keyrings, mems_allowed, mempolicy: */
385     spinlock_t         alloc_lock;
386
387     /* Protection of the PI data structures: */
388     raw_spinlock_t     pi_lock;
389
390     struct wake_q_node  wake_q;
391
392     #ifdef CONFIG_RT_MUTEXES
393     /* PI waiters blocked on a rt_mutex held by this task:
394     ↪  */
395     struct rb_root_cached    pi_waiters;
396     /* Updated under owner's pi_lock and rq lock */
397     struct task_struct       *pi_top_task;
398     /* Deadlock detection and priority inheritance
399     ↪  handling: */
400     struct rt_mutex_waiter   *pi_blocked_on;
401     #endif
402
403     #ifdef CONFIG_DEBUG_MUTEXES
404     /* Mutex deadlock detection: */
405     struct mutex_waiter      *blocked_on;
406     #endif
407
408     #ifdef CONFIG_DEBUG_ATOMIC_SLEEP
409     int                non_block_count;
410     #endif
411
412     #ifdef CONFIG_TRACE_IRQFLAGS
413     struct irqtrace_events    irqtrace;
414     unsigned int            hardirq_threaded;
415     u64                    hardirq_chain_key;
416     int                    softirqs_enabled;
417     int                    softirq_context;

```

```

415         int                irq_config;
416     #endif
417     #ifdef CONFIG_PREEMPT_RT
418         int                softirq_disable_cnt;
419     #endif
420
421     #ifdef CONFIG_LOCKDEP
422     # define MAX_LOCK_DEPTH          48UL
423         u64                curr_chain_key;
424         int                lockdep_depth;
425         unsigned int       lockdep_recursion;
426         struct held_lock    held_locks[MAX_LOCK_DEPTH];
427     #endif
428
429     #if defined(CONFIG_UBSAN) && !defined(CONFIG_UBSAN_TRAP)
430         unsigned int       in_ubsan;
431     #endif
432
433     /* Journalling filesystem info: */
434     void                *journal_info;
435
436     /* Stacked block device info: */
437     struct bio_list      *bio_list;
438
439     /* Stack plugging: */
440     struct blk_plug      *plug;
441
442     /* VM state: */
443     struct reclaim_state  *reclaim_state;
444
445     struct backing_dev_info  *backing_dev_info;
446
447     struct io_context     *io_context;
448
449     #ifdef CONFIG_COMPACTION
450         struct capture_control  *capture_control;
451     #endif
452     /* Ptrace state: */
453     unsigned long         ptrace_message;
454     kernel_siginfo_t      *last_siginfo;
455

```



```

456     struct task_io_accounting    ioac;
457 #ifdef CONFIG_PSI
458     /* Pressure stall state */
459     unsigned int                psi_flags;
460 #endif
461 #ifdef CONFIG_TASK_XACCT
462     /* Accumulated RSS usage: */
463     u64                          acct_rss_mem1;
464     /* Accumulated virtual memory usage: */
465     u64                          acct_vm_mem1;
466     /* stime + utime since last update: */
467     u64                          acct_timexpd;
468 #endif
469 #ifdef CONFIG_CPUSETS
470     /* Protected by ->alloc_lock: */
471     nodemask_t                  mems_allowed;
472     /* Sequence number to catch updates: */
473     seqcount_spinlock_t         mems_allowed_seq;
474     int                          cpuset_mem_spread_rotor;
475     int                          cpuset_slab_spread_rotor;
476 #endif
477 #ifdef CONFIG_CGROUPS
478     /* Control Group info protected by css_set_lock: */
479     struct css_set __rcu         *cgroups;
480     /* cg_list protected by css_set_lock and
481        ↪ tsk->alloc_lock: */
481     struct list_head            cg_list;
482 #endif
483 #ifdef CONFIG_X86_CPU_RESCTRL
484     u32                          closid;
485     u32                          rmid;
486 #endif
487 #ifdef CONFIG_FUTEX
488     struct robust_list_head __user *robust_list;
489 #ifdef CONFIG_COMPAT
490     struct compat_robust_list_head __user
491        ↪ *compat_robust_list;
491 #endif
492     struct list_head            pi_state_list;
493     struct futex_pi_state        *pi_state_cache;
494     struct mutex                 futex_exit_mutex;

```

```

495         unsigned int                futex_state;
496     #endif
497     #ifdef CONFIG_PERF_EVENTS
498         struct
499             ↪ perf_event_context      *perf_event_ctxp[perf_nr_task_contexts];
500         struct mutex                perf_event_mutex;
501         struct list_head            perf_event_list;
502     #endif
503     #ifdef CONFIG_DEBUG_PREEMPT
504         unsigned long                preempt_disable_ip;
505     #endif
506     #ifdef CONFIG_NUMA
507         /* Protected by alloc_lock: */
508         struct mempolicy             *mempolicy;
509         short                        il_prev;
510         short                        pref_node_fork;
511     #endif
512     #ifdef CONFIG_NUMA_BALANCING
513         int                          numa_scan_seq;
514         unsigned int                 numa_scan_period;
515         unsigned int                 numa_scan_period_max;
516         int                          numa_preferred_nid;
517         unsigned long                numa_migrate_retry;
518         /* Migration stamp: */
519         u64                          node_stamp;
520         u64                          last_task_numa_placement;
521         u64                          last_sum_exec_runtime;
522         struct callback_head          numa_work;
523
524         /*
525          * This pointer is only modified for current in
526          * ↪ syscall and
527          * pagefault context (and for tasks being destroyed),
528          * ↪ so it can be read
529          * from any of the following contexts:
530          * - RCU read-side critical section
531          * - current->numa_group from everywhere
532          * - task's runqueue locked, task not running
533          */
534         struct numa_group __rcu      *numa_group;

```

```

533     /*
534     * numa_faults is an array split into four regions:
535     * faults_memory, faults_cpu, faults_memory_buffer,
536     *   ↪ faults_cpu_buffer
537     * in this precise order.
538     *
539     * faults_memory: Exponential decaying average of
540     *   ↪ faults on a per-node
541     * basis. Scheduling placement decisions are made
542     *   ↪ based on these
543     * counts. The values remain static for the duration
544     *   ↪ of a PTE scan.
545     * faults_cpu: Track the nodes the process was running
546     *   ↪ on when a NUMA
547     * hinting fault was incurred.
548     * faults_memory_buffer and faults_cpu_buffer: Record
549     *   ↪ faults per node
550     * during the current scan window. When the scan
551     *   ↪ completes, the counts
552     * in faults_memory and faults_cpu decay and these
553     *   ↪ values are copied.
554     */
555     unsigned long          *numa_faults;
556     unsigned long          total_numa_faults;
557
558     /*
559     * numa_faults_locality tracks if faults recorded
560     *   ↪ during the last
561     * scan window were remote/local or failed to migrate.
562     *   ↪ The task scan
563     * period is adapted based on the locality of the
564     *   ↪ faults with different
565     * weights depending on whether they were shared or
566     *   ↪ private faults
567     */
568     unsigned long          numa_faults_locality[3];
569
570     unsigned long          numa_pages_migrated;
571 #endif /* CONFIG_NUMA_BALANCING */
572
573 #ifdef CONFIG_RSEQ

```

```

562     struct rseq __user *rseq;
563     u32 rseq_sig;
564     /*
565      * RmW on rseq_event_mask must be performed atomically
566      * with respect to preemption.
567      */
568     unsigned long rseq_event_mask;
569 #endif
570
571     struct tlbflush_unmap_batch    tlb_ubic;
572
573     union {
574         refcount_t                rcu_users;
575         struct rcu_head            rcu;
576     };
577
578     /* Cache last used pipe for splice(): */
579     struct pipe_inode_info         *splice_pipe;
580
581     struct page_frag              task_frag;
582
583 #ifdef CONFIG_TASK_DELAY_ACCT
584     struct task_delay_info         *delays;
585 #endif
586
587 #ifdef CONFIG_FAULT_INJECTION
588     int                            make_it_fail;
589     unsigned int                   fail_nth;
590 #endif
591     /*
592      * When (nr_dirtied >= nr_dirtied_pause), it's time to
593      ↪ call
594      * balance_dirty_pages() for a dirty throttling pause:
595      */
596     int                            nr_dirtied;
597     int                            nr_dirtied_pause;
598     /* Start of a write-and-pause period: */
599     unsigned long                  dirty_paused_when;
600
601 #ifdef CONFIG_LATENCYTOP
602     int                            latency_record_count;

```

```

602     struct
        ↳ latency_record          latency_record[LT_SAVECOUNT];
603 #endif
604     /*
605      * Time slack values; these are used to round up
        ↳ poll() and
606      * select() etc timeout values. These are in
        ↳ nanoseconds.
607      */
608     u64          timer_slack_ns;
609     u64          default_timer_slack_ns;
610
611 #if defined(CONFIG_KASAN_GENERIC) ||
        ↳ defined(CONFIG_KASAN_SW_TAGS)
612     unsigned int      kasan_depth;
613 #endif
614
615 #ifdef CONFIG_KCSAN
616     struct kcsan_ctx      kcsan_ctx;
617 #ifdef CONFIG_TRACE_IRQFLAGS
618     struct irqtrace_events      kcsan_save_irqtrace;
619 #endif
620 #endif
621
622 #if IS_ENABLED(CONFIG_KUNIT)
623     struct kunit          *kunit_test;
624 #endif
625
626 #ifdef CONFIG_FUNCTION_GRAPH_TRACER
627     /* Index of current stored address in ret_stack: */
628     int          curr_ret_stack;
629     int          curr_ret_depth;
630
631     /* Stack of return addresses for return function
        ↳ tracing: */
632     struct ftrace_ret_stack      *ret_stack;
633
634     /* Timestamp for last schedule: */
635     unsigned long long      ftrace_timestamp;
636
637     /*

```

```

638     * Number of functions that haven't been traced
639     * because of depth overrun:
640     */
641     atomic_t          trace_overrun;
642
643     /* Pause tracing: */
644     atomic_t          tracing_graph_pause;
645 #endif
646
647 #ifdef CONFIG_TRACING
648     /* State flags for use by tracers: */
649     unsigned long      trace;
650
651     /* Bitmask and counter of trace recursion: */
652     unsigned long      trace_recursion;
653 #endif /* CONFIG_TRACING */
654
655 #ifdef CONFIG_KCOV
656     /* See kernel/kcov.c for more details. */
657
658     /* Coverage collection mode enabled for this task (0
659      ↪ if disabled): */
659     unsigned int        kcov_mode;
660
661     /* Size of the kcov_area: */
662     unsigned int        kcov_size;
663
664     /* Buffer for coverage collection: */
665     void                *kcov_area;
666
667     /* KCOV descriptor wired with this task or NULL: */
668     struct kcov          *kcov;
669
670     /* KCOV common handle for remote coverage collection:
671      ↪ */
671     u64                 kcov_handle;
672
673     /* KCOV sequence number: */
674     int                 kcov_sequence;
675
676     /* Collect coverage from softirq context: */

```

```

677         unsigned int                kcov_softirq;
678     #endif
679
680     #ifdef CONFIG_MEMCG
681         struct mem_cgroup            *memcg_in_oom;
682         gfp_t                        memcg_oom_gfp_mask;
683         int                          memcg_oom_order;
684
685         /* Number of pages to reclaim on returning to
        ↪ userland: */
686         unsigned int                memcg_nr_pages_over_high;
687
688         /* Used by memcontrol for targeted memcg charge: */
689         struct mem_cgroup            *active_memcg;
690     #endif
691
692     #ifdef CONFIG_BLK_CGROUP
693         struct request_queue         *throttle_queue;
694     #endif
695
696     #ifdef CONFIG_UPROBES
697         struct uprobe_task           *utask;
698     #endif
699     #if defined(CONFIG_BCACHE) ||
    ↪ defined(CONFIG_BCACHE_MODULE)
700         unsigned int                sequential_io;
701         unsigned int                sequential_io_avg;
702     #endif
703         struct kmap_ctrl             kmap_ctrl;
704     #ifdef CONFIG_DEBUG_ATOMIC_SLEEP
705         unsigned long                task_state_change;
706     # ifdef CONFIG_PREEMPT_RT
707         unsigned long                saved_state_change;
708     # endif
709     #endif
710         int                          pagefault_disabled;
711     #ifdef CONFIG_MMU
712         struct task_struct           *oom_reaper_list;
713     #endif
714     #ifdef CONFIG_VMAP_STACK
715         struct vm_struct             *stack_vm_area;

```

```

716 #endif
717 #ifdef CONFIG_THREAD_INFO_IN_TASK
718     /* A live task holds one reference: */
719     refcount_t          stack_refcount;
720 #endif
721 #ifdef CONFIG_LIVEPATCH
722     int patch_state;
723 #endif
724 #ifdef CONFIG_SECURITY
725     /* Used by LSM modules for access restriction: */
726     void                *security;
727 #endif
728 #ifdef CONFIG_BPF_SYSCALL
729     /* Used by BPF task local storage */
730     struct bpf_local_storage __rcu    *bpf_storage;
731     /* Used for BPF run context */
732     struct bpf_run_ctx      *bpf_ctx;
733 #endif
734
735 #ifdef CONFIG_GCC_PLUGIN_STACKLEAK
736     unsigned long          lowest_stack;
737     unsigned long          prev_lowest_stack;
738 #endif
739
740 #ifdef CONFIG_X86_MCE
741     void __user            *mce_vaddr;
742     __u64                  mce_kflags;
743     u64                    mce_addr;
744     __u64                  mce_ripv : 1,
745                             mce_whole_page : 1,
746                             __mce_reserved : 62;
747     struct callback_head    mce_kill_me;
748     int                    mce_count;
749 #endif
750
751 #ifdef CONFIG_KRETPROBES
752     struct llist_head      kretprobe_instances;
753 #endif
754
755 #ifdef CONFIG_ARCH_HAS_PARANOID_L1D_FLUSH
756     /*

```



```

757     * If L1D flush is supported on mm context switch
758     * then we use this callback head to queue kill work
759     * to kill tasks that are not running on SMT disabled
760     * cores
761     */
762     struct callback_head      lld_flush_kill;
763 #endif
764
765     /*
766     * New fields for task_struct should be added above
767     ↪ here, so that
768     * they are included in the randomized portion of
769     ↪ task_struct.
770     */
771     randomized_struct_fields_end
772
773     /* CPU-specific state of this task: */
774     struct thread_struct      thread;
775
776     /*
777     * WARNING: on x86, 'thread_struct' contains a
778     ↪ variable-sized
779     * structure. It *MUST* be at the end of
780     ↪ 'task_struct'.
781     *
782     * Do not put anything below here!
783     */
784 };

```