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

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Мониторинг параметров планирования процесса»

Студент **ИУ7-75Б** _____ **В. А. Лебедев**

(Подпись, дата) (И.О.Фамилия)

Руководитель _____ **Н. Ю. Рязанова**

(Подпись, дата) (И.О.Фамилия)

2025 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)
« _____ » _____ 2024 г.

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине Операционные системы

Студент группы ИУ7-75Б

Лебедев Владимир Александрович
(Фамилия, имя, отчество)

Тема курсовой работы Мониторинг параметров планирования процесса

Направленность КР (учебная, исследовательская, практическая, производственная, др.): Учебная
Источник тематики (кафедра, предприятие, НИР): Кафедра

График выполнения работы: 25% к _4_ нед., 50% к _7_ нед., 75% к _11_ нед., 100% к _14_ нед.

Задание
Разработать загружаемый модуль ядра для получения информации о планировании процесса в системе: класс планирования процесса, изменения в дереве задач планировщика, время ожидания процесса в очереди, проверки необходимости вытеснения, смена класса планирования, вес процесса, данные из структуры sched_statistics.

Оформление курсовой работы:
Расчетно-пояснительная записка на 12-32 листах формата А4.

Дата выдачи задания « ___ » _____ 20__ г.

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

Студент _____ (Подпись, дата) В. А. Лебедев (И.О.Фамилия)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **5**

1 Аналитический раздел **6**

1.1 Постановка задачи 6

1.2 Анализ структур ядра, связанных с работой планировщика 6

1.2.1 Структура task_struct 6

1.2.2 Структуры, хранящие статистику планирования 11

1.2.3 Структура sched_class 12

1.2.4 Структуры очередей 14

1.3 Классы планировщиков 21

1.4 Способы перехвата функций ядра 25

1.4.1 Модификация таблицы системных вызовов 25

1.4.2 Linux Security API 25

1.4.3 kprobes 25

1.4.4 ftrace 26

1.4.5 Сравнение способов перехвата функций ядра 26

2 Конструкторский раздел **29**

2.1 Последовательность действий 29

2.2 Разработка алгоритма работы предобработчика 30

2.3 Разработка алгоритма создания элемента списка 31

2.4 Структура программного обеспечения 32

3 Технологический раздел **33**

3.1 Выбор языка и среды программирования 33

3.2 Реализация структур для хранения информации о планировании 33

3.3 Реализация создания элемента списка 35

3.4 Реализация обработчика функции task_tick_fair 37

4 Исследовательский раздел **38**

4.1 Демонстрация работы программы 38

ЗАКЛЮЧЕНИЕ **40**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **42**

ПРИЛОЖЕНИЕ А **43**

ВВЕДЕНИЕ

Планирование — это выбор процесса из очереди готовых к выполнению процессов для распределения ресурсов процессора в соотвествии некоторым алгоритмом. Целью планирования процессов является максимизация времени работы процессора [1].

Целью данной работы является разработка загружаемого модуля ядра для получения информации о параметрах планирования процесса.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу необходимо разработать загружаемый модуль ядра для получения информации о планировании процесса в системе: классе планирования и его смене, проверках необходимости вытеснения, времени ожидания в очереди, изменениях в дереве задач планировщика, данных из структуры *sched_statistics*. Для решения поставленной задачи необходимо:

- провести анализ структур и функций, предоставляющих возможность реализовать поставленную задачу;
- провести анализ способов перехвата функций;
- разработать алгоритмы и структуру загружаемого модуля, обеспечивающего отслеживание работы планировщика;
- спроектировать и реализовать загружаемый модуль ядра;
- проанализировать работу загружаемого модуля ядра.

1.2 Анализ структур ядра, связанных с работой планировщика

1.2.1 Структура task_struct

Структура *task_struct*, называемая дескриптором процесса, описывает процесс в ядре. Определение этой структуры находится в заголовочном файле <linux/sched.h>. В листинге 1 представлены поля структуры *task_struct*, необходимые для выполнения поставленной задачи.

Листинг 1 – Структура task_struct

```
1  struct task_struct {
2      ...
3      int                prio;
4      int                static_prio;
5      int                normal_prio;
6      unsigned int               rt_priority;
7      ...
8      unsigned int               policy;
9      ...
10     struct sched_entity      se;
11     struct sched_rt_entity       rt;
12     struct sched_dl_entity       dl;
13     ...
14     const struct sched_class     *sched_class;
15     ...
16     struct sched_statistics          stats;
17     ...
18     struct sched_info        sched_info;
19     ...
20     pid_t                  pid;
21     ...
22 }
```

Приведенные поля описаны ниже:

1. *prio* — эффективный приоритет процесса, использующийся при принятии решений при планировании. Его значение варьируется от 0 до 139, причем меньшее значение соответствует более высокому приоритету, и зависит от *rt_priority* для процессов реального времени и от *static_prio* для остальных процессов. *prio* позволяет сопоставить диапазоны приоритетов разных классов планирования в один диапазон, состоящий из приоритета реального времени (от 0 до 99) и приоритета остальных процессов (от 100 до 139). Значения от 100 до 139 получаются

с учетом значения *nice* — числа в диапазоне от -20 до 19, показывающего, насколько процесс «вежлив» по отношению к другим процессам. Чем выше значение *nice*, тем ниже приоритет процесса. По умолчанию значение *nice* равно 0. Приоритеты процессов UNIX представлены на рисунке 1.

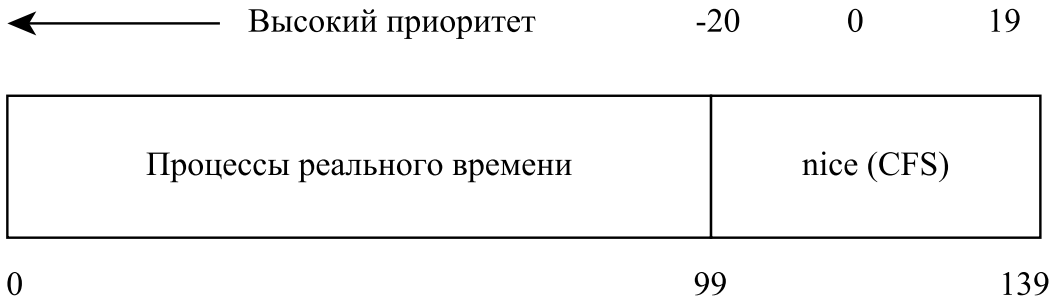

Рисунок 1 – Приоритеты процессов UNIX

Для процессов с дисциплиной планирования SCHED_DEADLINE *prio* равно -1, так как для них данное поле не учитывается при планировании.

2. *static_prio* — приоритет процессов, не являющихся процессами реального времени. Значение этого поля рассчитывается по формуле $static\_prio = 120 + nice$. При отсутствии временного повышения приоритета со стороны ядра значения *static_prio* и *prio*. Для процессов реального времени данное поле не учитывается.
3. *normal_prio* — значение, равное приоритету процесса без временного повышения со стороны ядра. Для процессов реального времени *normal_prio* находится по формуле $MAX\_RT\_PRIO - 1 - rt\_priority$, а для остальных процессов оно равно *static_prio*. Основная цель *normal_prio* — предотвратить голодание процессора для других процессов, вызванное дочерними процессами одного или нескольких процессов с повышенными приоритетами.
4. *rt_priority* — приоритет процесса реального времени. Диапазон при-

оитетов реального времени сопоставляется с внутренним диапазоном приоритетов по формуле $MAX\_RT\_PRIO - 1 - rt\_priority$, где $MAX\_RT\_PRIO = 100$. Таким образом, $rt\_priority$ от 0 до 99 сопоставляется со значением внутреннего приоритета от 99 до 0. Чем больше $rt\_priority$, тем выше приоритет процесса. $rt\_priority$ влияет на приоритет только процессов реального времени и игнорируется для остальных процессов.

5. *policy* — дисциплина планировая. Поле *policy* имеет одно из следующих значений: SCHED_NORMAL, SCHED_FIFO, SCHED_RR, SCHED_BATCH, SCHED_IDLE, SCHED_DEADLINE.
6. *se* — структура, содержащая информацию, необходимую планировщику CFS.
7. *rt* — структура, содержащая информацию, необходимую для планирования процессов с дисуиплиной планирования SCHED_FIFO или SCHED_RR.
8. *rt* — структура, содержащая информацию, необходимую для планирования процессов с дисциплиной планирования SCHED_DEADLINE.
9. *sched_class* — указатель на структуру, определяющую класс планировщика процесса.
10. *stats* — статистика планирования процесса.
11. *sched_info* — дополнительная информация о планировании процесса.
12. *pid* — идентификатор процесса.

Параметры, влияющие на планирование, устанавливаются с помощью функции *sched_setattr*, прототип которой приведен в листинге 2.

Листинг 2 – Прототип функции sched_setattr

```
1 int sched_setattr(struct task_struct *p, const struct sched_attr
    *attr);
```

Параметры передаются с использованием структуры *sched_attr*, неко-

торые поля которой приведены в листинге 3.

Листинг 3 – Структура sched_attr

```
1  struct sched_attr {
2      ...
3      __u32 sched_policy;
4      ...
5      /* SCHED_NORMAL, SCHED_BATCH */
6      __s32 sched_nice;
7
8      /* SCHED_FIFO, SCHED_RR */
9      __u32 sched_priority;
10
11     /* SCHED_DEADLINE */
12     __u64 sched_runtime;
13     __u64 sched_deadline;
14     __u64 sched_period;
15     ...
16 };
```

Поле *sched_policy* содержит устанавливаемую дисциплину планирования. Поле *sched_nice* используется для процессов с дисциплиной планирования SCHED_NORMAL или SCHED_BATCH. В нем указывается новое значение nice, влияющее на приоритет процесса. Поле *sched_priority* используется для указания приоритета процесса с дисциплиной планирования SCHED_FIFO или SCHED_RR. Поля *sched_runtime*, *sched_deadline*, *sched_period* используются для установки параметров для процессов с дисциплиной планирования SCHED_DEADLINE.

1.2.2 Структуры, хранящие статистику планирования

Структуры *sched_statistics* и *sched_info* содержат в себе статистику, обновляемую в процессе работы планировщика. В листинге 4 приведены некоторые поля структуры *sched_statistics*.

Листинг 4 – Структура sched_statistics

```
1  struct sched_statistics {
2  #ifdef CONFIG_SCHEDSTATS
3      u64                 wait_start;
4      u64                 wait_max;
5      u64                 wait_count;
6      u64                 wait_sum;
7      u64                 iowait_count;
8      u64                 iowait_sum;
9
10     u64                 sleep_start;
11     u64                 sleep_max;
12     s64                 sum_sleep_runtime;
13
14     u64                 block_start;
15     u64                 block_max;
16     s64                 sum_block_runtime;
17
18     s64                 exec_max;
19     u64                 slice_max;
20     ...
21 #endif /* CONFIG_SCHEDSTATS */
22 } ____cacheline_aligned;
```

Поля *wait_start*, *wait_max*, *wait_count* и *wait_sum* отражают статистику о времени ожидания процессом очереди на выполнение, поля *iowait_count* и *iowait_sum* — статистику времени ожидания завершения

операций ввода/вывода. Поля *sleep_start*, *sleep_max* и *sum_sleep_runtime* содержат информацию о времени, проведенном в состоянии сна, а поля *block_start*, *block_max* и *sum_block_runtime* — информацию о времени, проведенном в состоянии блокировки. *exec_max* отражает максимальное время выполнения без вытеснения, *slice_max* — максимальный размер кванта времени, выделенный процессу.

В листинге 5 приведены поля структуры *sched_info*.

Листинг 5 – Структура sched_info

```
1 struct sched_info {
2 #ifdef CONFIG_SCHED_INFO
3     unsigned long           pcount;
4     unsigned long long      run_delay;
5     unsigned long long      last_arrival;
6     unsigned long long      last_queued;
7 #endif /* CONFIG_SCHED_INFO */
8 };
```

Описание приведенных полей представлено ниже:

1. *pcount* — число раз, которое процесс выполнялся на данном ядре.
2. *run_delay* — суммарное время, проведенное в ожидании очереди на выполнение.
3. *last_arrival* — время, когда процесс последний раз начал выполняться.
4. *last_queued* — время, когда процесс последний раз был поставлен в очередь на выполнение.

1.2.3 Структура sched_class

Структура *sched_class* предоставляет интерфейс для работы с планировщиками. Данная структура содержит указатели на функции, реализуемые для каждого планировщика. В листинге 6 приведены поля структуры

sched_class, необходимые для выполнения поставленной задачи.

Листинг 6 – Структура sched_class

```
1  struct sched_class {
2      ...
3      void (*enqueue_task) (struct rq *rq, struct task_struct *p,
           int flags);
4      void (*dequeue_task) (struct rq *rq, struct task_struct *p,
           int flags);
5      void (*yield_task)   (struct rq *rq);
6      bool (*yield_to_task)(struct rq *rq, struct task_struct *p);
7      void (*wakeup_preempt)(struct rq *rq, struct task_struct *p,
           int flags);
8      struct task_struct *(*pick_next_task)(struct rq *rq);
9      void (*put_prev_task)(struct rq *rq, struct task_struct *p);
10     void (*set_next_task)(struct rq *rq, struct task_struct *p,
           bool first);
11     ...
12     void (*task_tick)(struct rq *rq, struct task_struct *p, int
           queued);
13     ...
14     void (*prio_changed) (struct rq *this_rq, struct task_struct
           *task, int oldprio);
15     ...
16     void (*update_curr)(struct rq *rq);
17     ...
18 };
```

Назначение функций планировщика представлено ниже:

1. *enqueue_task* — добавление процесса в очередь;
2. *dequeue_task* — удаление процесса из очереди;
3. *yield_task* — позволяет процессу добровольно уступить процессор;
4. *yield_to_task* — позволяет процессу добровольно уступить процессор определенному процессу;

5. *wakeup_preempt* — обработка пробуждения процесса и возможное вытеснение текущего выполняющегося процесса;
6. *pick_next_task* — выбор следующего процесса для выполнения из очереди rq;
7. *put_prev_task* — вызывается после того, как процесс перестал выполняться;
8. *set_next_task* — вызывается после того, как процесс сменил класс планирования, группу или был запланирован на выполнение;
9. *task_tick* — вызывается функцией *scheduler_tick*, которая вызывается ядром с частотой, соответствующей прерыванию от системного таймера;
10. *prio_changed* — обработка изменения приоритета процесса;
11. *update_curr* — обновление информации о процессе, например *vruntime*.

Класс планировщика может быть установлен с помощью функции *sched_setscheduler*, прототип которой приведен в листинге 7.

Листинг 7 – Прототип функции sched_setscheduler

```
1 int sched_setscheduler(struct task_struct *p, int policy, const
    struct sched_param *param);
```

Структура *sched_param* содержит единственное поле *sched_priority*, использующееся для процессов с дисциплиной планирования SCHED_FIFO или SCHED_RR.

1.2.4 Структуры очередей

Для представления очереди к процессору используется структура *rq* (runqueue). В листинге 8 представлены поля структуры *rq*, необходимые для выполнения поставленной задачи.

Листинг 8 – Структура rq

```
1 struct rq {
2     ...
3     struct cfs_rq cfs;
4     struct rt_rq rt;
5     struct dl_rq dl;
6     ...
7     struct task_struct __rcu    *curr;
8     ...
9 }
```

runqueue содержит очереди для процессов с классами планирования *RT*, *Deadline* и *CFS*. Поле *curr* содержит указатель на процесс, выполняемый на процессоре, к которому относится данная очередь.

Некоторые поля структуры cfs_rq, представляющей очередь для класса планирования CFS, приведены в листинге 9.

Листинг 9 – Структура cfs_rq

```
1  struct cfs_rq {
2      struct load_weight  load;
3      unsigned int        nr_running;
4      ...
5      unsigned int        idle_nr_running;    /* SCHED_IDLE */
6      ...
7      u64         min_vruntime;
8      struct rb_root_cached   tasks_timeline;
9      struct sched_entity *curr;
10     struct sched_entity *next;
11     ...
12 }
```

Описание приведенных полей представлено ниже:

1. *load* — суммарный «вес» процессов очереди. Структура *load_weight* содержит поля *weight* и *inv_weight* для суммарного «веса» процессов

в очереди и обратного значения.

2. *nr_running* — количество процессов в очереди, находящихся в состоянии running или runnable.
3. *idle_nr_running* — количество процессов в очереди, имеющих дисциплину планирования SCHED_IDLE.
4. *min_vruntime* — минимальное значение vruntime среди процессов в очереди. Значение *min_vruntime* присваивается в качестве vruntime добавляемых в очередь процессов.
5. *task_timeline* — красно-черное дерево, содержащее процессы, готовые к выполнению.
6. *curr* — указатель на информацию о текущем выполняемом процессе. Если текущий выполняемый процесс имеет класс планирования CFS, то данное поле содержит информацию о процессе, указатель на который содержится в поле *curr* структуры *rq*.
7. *next* — указатель на информацию о процессе, который будет выполняться следующим.

В листинге 10 приведена структура *rb_root_cached*.

Листинг 10 – Структура rb_root_cached

```
1 struct rb_root_cached {
2     struct rb_root rb_root;
3     struct rb_node *rb_leftmost;
4 };
```

Поле *rb_root* содержит информацию о корне дерева, которая содержит указатель на корень. Поле *rb_leftmost* содержит указатель на самый левый узел дерева, имеющий минимальное значение ключа. Для CFS ключом является значение vruntime.

В листинге 11 приведены некоторые поля структуры *sched_entity*, содержащей информацию о процессе для планирования в рамках CFS.

Листинг 11 – Структура sched_entity

```
1  struct sched_entity {
2      struct load_weight      load;
3      struct rb_node          run_node;
4      ...
5      unsigned int            on_rq;
6      ...
7      u64             vruntime;
8      s64             vlag;
9      u64             slice;
10     ...
11 }
```

Описание приведенных полей представлено ниже:

1. *load* — «вес» процесса, определяющий, какую долю процессорного времени должен получить данный процесс по сравнению с другими.
2. *run_node* — узел для вставки в красно-черное дерево *task_timeline* в *cfs_rq*.
3. *on_rq* — флаг, показывающий, находится ли данный процесс в очереди.
4. *vruntime* — виртуальное время выполнения процесса, которое представляет собой фактическое время выполнения (количество времени, затраченного на выполнение), нормированное (или взвешенное) количеством выполняемых процессов.
5. *vlag* — смещение vruntime при пробуждении процесса.
6. *slice* — квант времени, выделенный процессу.

В листинге 12 приведены некоторые поля структуры *rt_rq*, представляющей очередь для класса планирования процессов реального времени.

Листинг 12 – Структура rt_rq

```
1 struct rt_rq {
2     struct rt_prio_array     active;
3     unsigned int         rt_nr_running;
4     ...
5 }
```

Поле *rt_nr_running* содержит количество процессов реального времени, находящихся в этой очереди и находящихся в состоянии running или runnable. Поле *active* содержит структуру, описывающую массив приоритетных списков, содержащих процессы реального в зависимости от их приоритета.

В листинге 13 приведены некоторые поля структуры *sched_rt_entity*, содержащую информацию для планирования процессов реального времени.

Листинг 13 – Структура sched_rt_entity

```
1  struct sched_rt_entity {
2      struct list_head         run_list;
3      unsigned long            timeout;
4      ...
5      unsigned int             time_slice;
6      unsigned short           on_rq;
7      unsigned short           on_list;
8      struct sched_rt_entity       *back;
9      ...
10 }
```

Описание представленных полей приведено ниже:

1. *run_list* — узел для вставки в приоритетный список процессов, готовых к выполнению.
2. *timeout* — время, когда истечет выделенный процессу квант времени. Используется для процессов с дисциплиной планирования SCHED_RR.
3. *time_slice* — квант времени, выделенный процессу с дисциплиной пла-

нирования SCHED_RR.

4. *on_rq* — флаг, показывающий, находится ли процесс в очереди на выполнение.
5. *on_list* — флаг, показывающий, находится ли процесс в приоритетном списке.
6. *back* — указатель на предыдущий элемент в списке.

В листинге 14 приведены некоторые поля структуры *dl_rq*, представляющей очередь для процессов, имеющих дисциплину планирования SCHED_DEADLINE.

Листинг 14 – Структура dl_rq

```
1 struct dl_rq {
2     struct rb_root_cached   root;
3     unsigned int            dl_nr_running;
4     ...
5 }
```

Поле *root* содержит информацию о корне и крайнем левом элементе красно-черного дерева deadline-планировщика. Поле *dl_nr_running* содержит количество процессов в очереди, находящихся в состоянии running или runnable.

В листинге 15 приведены некоторые поля структуры *sched_dl_entity*, содержащей информацию о процессе для работы deadline-планировщика.

Листинг 15 – Структура sched_dl_entity

```
1 struct sched_dl_entity {
2     struct rb_node          rb_node;
3     u64                     dl_runtime;
4     u64                     dl_deadline;
5     u64                     dl_period;
6     u64                     dl_bw;
7     u64                     dl_density;
8     s64                     runtime;
```

```
9       u64                 deadline;
10      ...
11  }
```

Описание приведенных полей представлено ниже:

1. *run_node* — узел для вставки в красно-черное дерево в *dl_rq*.
2. *dl_runtime* — максимальное время выполнения процесса в микросекундах.
3. *dl_deadline* — относительный крайний срок завершения процесса.
4. *dl_period* — период между двумя процессами.
5. *dl_bw* — пропускная способность, равная частному *dl_runtime* и *dl_period*.
6. *dl_density* — плотность, равная частному *dl_runtime* и *dl_deadline*.
7. *runtime* — оставшееся время для выполнения процесса.
8. *deadline* — абсолютный крайний срок заверщения выполнения процесса, вычисляемый суммированием текущего времени и *dl_deadline*.

Поля *dl_runtime*, *dl_deadline* и *dl_period* устанавливаются при вызове *sched_setaddr* и остаются неизменными до следующего вызова *sched_setaddr*.

Связь очередей и огранизация хранения информации о процессах представлены на рисунке 2.

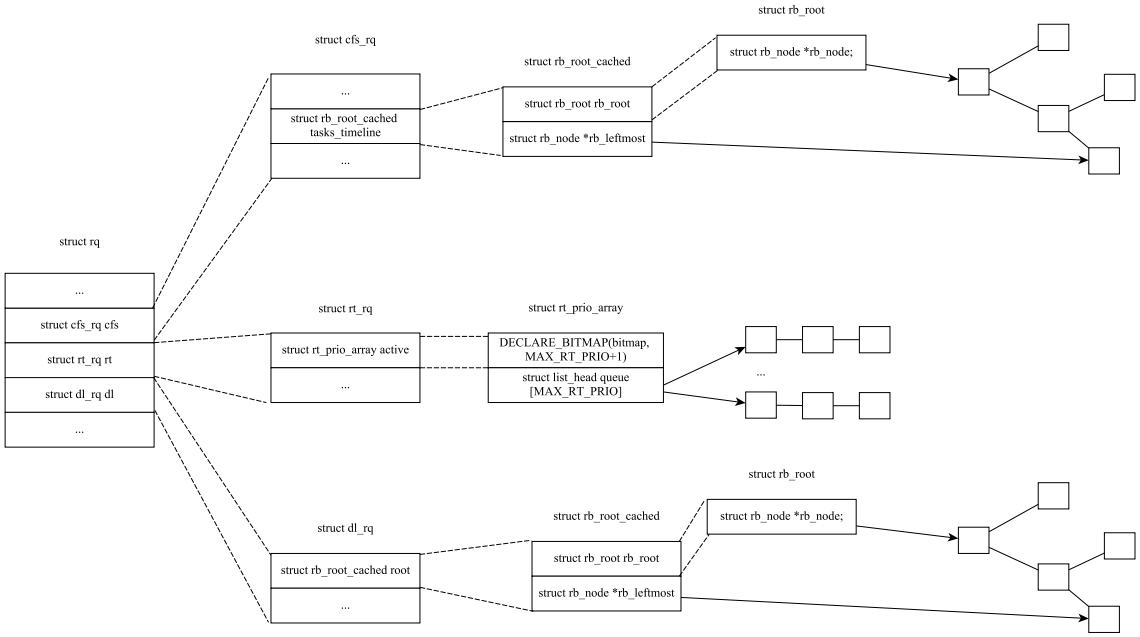

Рисунок 2 – Связь очередей процессов, готовых к выполнению

1.3 Классы планировщиков

Существует пять классов планировщиков, объявленных в заголовочном файле <sched.h>:

1. *stop_sched_class*;
2. *dl_sched_class*;
3. *rt_sched_class*;
4. *fair_sched_class*;
5. *idle_sched_class*.

Для каждого класса реализованы функции, указатели на которые хранятся в структуре *sched_class*. В приведенном выше списке классы планировщиков расположены в порядке уменьшения их приоритета. Класс планировщика с более высоким приоритетом может вытеснить процесс с классом планировщика более низкого приоритета.

Stop-класс планировщика используется для внутреннего управления процессором в случае необходимости остановки его работы. Idle-класс пла-

нировщика используется для idle-потока для каждого процессора, когда нет других процессов для планирования [1].

Deadline-планировщик используется для процессов с дисциплиной планирования SCHED_DEADLINE. Функции для его работы реализованы в файле <kernel/sched/deadline.c>. Для планирования процессов данный планировщик использует три значения: runtime, deadline и period, которые хранятся в полях *dl_runtime*, *dl_deadline* и *dl_period* в структуре *sched_dl_entity* для процесса. Процесс должен получать количество микросекунд, равное runtime, на выполнение через каждый промежуток времени, равный period, и быть выполнен в течение промежутка времени, равного deadline от начала периода. Планировщик поддерживает красно-черное дерево процессов, в котором ключом является значение deadline. Для выполнения выбирается процесс с наименьшим значением ключа. Каждый раз, когда процесс выходит из состояния сна, планировщик вычисляет «крайний срок планирования», используя алгоритм CBS (Constant Bandwidth Server). Состояние процесса описывается понятиями запланированного крайнего срока (scheduling deadline) и оставшегося времени выполнения (remaining runtime), которые изначально равны нулю. Когда процесс становится готовым к выполнению, проверяется условие, приведенное в формуле (1).

$$\frac{\text{remaining runtime}}{\text{scheduling deadline} - \text{current time}} > \frac{runtime}{period} \tag{1}$$

Если условие истинно или запланированный крайний срок меньше текущего времени, запланированный крайний срок становится равным сумме текущего времени и величины deadline, а оставшееся время выполнения инициализируется значением runtime, иначе эти значения остаются без изменений. Если процесс выполнялся t микросекунд, то оставшееся время выполнения уменьшается на t. Технически оставшееся время выполнения уменьшается при прерывании от системного таймера, при вытеснении процесса или при

его удалении из очереди процессов, готовых к выполнению. Когда оставшееся время выполнения становится меньше нуля, процесс переходит в «истощенное» состояние (throttled/depleted). Время «пополнения» (replenishment time) устанавливается равным текущему значению запланированного крайнего срока, при наступлении которого к запланированному крайнему сроку добавляется значение period, а к оставшемуся времени выполнения — значение runtime. Нет ограничений на то, какие процессы могут использовать данную дисциплину планирования, однако, как указано в документации ядра, она особенно подходит для периодических или спорадических процессов реального времени [2].

Real-time (RT) планировщик используется для процессов реального времени с дисциплиной планирования SCHED_FIFO или SCHED_RR. Функции для его работы реализованы в файле <kernel/sched/rt.c>. Для процессов с дисциплиной планирования SCHED_FIFO не происходит квантования времени. Когда процесс готов к выполнению, он помещается в конец очереди процессов с соответствующим приоритетом. Процесс может быть вытеснен процессом с более высоким приоритетом, при этом оставаясь в начале очереди процессов с соответствующим ему приоритетом, и может продолжить выполнение, когда закончатся процессы с большим приоритетом. Процесс может переместиться в конец очереди только при вызовах *sched_setscheduler*, *sched_setattr* или *sched_yield*. Процессы с дисциплиной планирования SCHED_RR могут выполняться в течение выделенного кванта времени, по истечение которого будут помещены в конец очереди с соответствующим приоритетом [1].

Планировщик CFS (Completely Fair scheduler) используется для процессов с дисциплиной планирования SCHED_NORMAL, SCHED_BATCH или SCHED_IDLE. Реализация функций для его работы приведена в файле <kernel/sched/fair.c>. Процессы с дисциплиной планирования SCHED_BATCH не могут вытеснять другие процессы и используются для

обработки пакетов в фоновом режиме. Их приоритет ниже, чем у процессов с дисциплиной планирования SCHED_NORMAL. SCHED_IDLE используется для процессов с наименьшим приоритетом, значение nice у них больше 19. CFS выдает каждому процессу $\frac{1}{n}$ времени процессора, где n — число процессов, как если бы процесс использовал соответствующую часть мощности процессора. CFS использует значение nice для определения приоритета процесса. Чтобы рассчитать фактический временной интервал, CFS ставит целью аппроксимацию бесконечно малой продолжительности планирования в идеальной многозадачности. Эта цель называется целевой задержкой (target latency). CFS накладывает ограничение на квант времени, выдаваемый процессу во избежание частых переключений контекста при приближении числа выполняемых процессов к бесконечности. Это ограничение называется минимальной гранулярностью (minimum granularity) [3]. Для хранения процессов планировщик CFS использует красно-черное дерево, в качестве ключа используется значение vruntime. Значение vruntime вычисляется по формуле (2).

$$vruntime = (\text{actual runtime}) \cdot \frac{1024}{weight} \tag{2}$$

«Вес» пропорционален приоритету, поэтому виртуальное время процесса с более высоким приоритетом растет медленнее, чем у процесса с более низким приоритетом. Тем самым достигается перемещение более приоритетных процессов в левую часть дерева планировщика. Рассчет значения vruntime происходит в функции *update_curr* [4].

1.4 Способы перехвата функций ядра

1.4.1 Модификация таблицы системных вызовов

Обработчики системных вызовов хранятся в таблице *sys_call_table*. Для добавления собственного обработчика необходимо сохранить старое значение обработчика и подставить в таблицу собственный обработчик. Недостатками данного подхода являются ограниченное число функций ядра, доступных для перехвата, и необходимость пересборки ядра [5].

1.4.2 Linux Security API

Linux Security API — специальный интерфейс, предоставляющий перехватчики функций ядра. В критических местах кода ядра расположены security-функции, вызывающие обратные вызовы, установленные security-модулем Security-модули являются частью ядра и требуют его пересборки для загрузки. В системе может быть только один security-модуль [5].

1.4.3 kprobes

kprobes — специальный интерфейс в ядре для установки обработчиков функций ядра для сбора информации. В обработчике имеется доступ к регистрам, позволяющий получить параметры функции и модифицировать их. Обработчики могут быть установлены до и после вызова функции. При регистрации обработчика, kprobes создает копию перехватываемой функции и заменяет ее первый байт инструкцией точки останова (int3 для x86_64). При

переходе на точку останова сохраняются регистры, а управление передается kprobes с помощью механизма notifier_call_chain и происходит переход к копии функции. Для извлечения аргументов необходимо знать, в каких регистрах они находятся. Надстройка jprobes позволяла решить эту проблему, однако она является устаревшей и удалена из современных версий ядра [6].

1.4.4 ftrace

ftrace — фреймворк в ядре, используемый для отладки и анализа. ftrace вставляет вызов $\_\_fentry()\_\_$ или $mcount()$ в начало перехватываемой функции с помощью ключей $-pg$ и $-mfentry$, указываемых при компиляции. Для оптимизации ftrace используется динамически: ядру известно расположение вызовов $\_\_fentry()\_\_$ и $mcount()$, поэтому на ранних этапах загрузки их код заменяется на nop. При включении трассирования в нужные функции вызовы ftrace добавляются обратно [5].

1.4.5 Сравнение способов перехвата функций ядра

Для сравнения способов перехвата функций ядра были выделены следующие критерии:

— Необходимость перекомпиляции ядра;

— Возможность перехвата любых функций ядра;

— Поддержка современными версиями ядра (начиная с версии 6.0).

Результаты сравнения способов перехвата функций ядра по выделенным критериям приведены в таблице 1.

Таблица 1 – Сравнение способов перехвата функций ядра

Способ перехвата	**Необходимость перекомпиляции ядра**	**Перехват любых функций ядра**	**Поддержка современными версиями ядра**
Перехват через таблицу системных вызовов	Да	Нет	Нет
LSM	Да	Нет	Да
kprobes	Нет	Да	Да
ftrace	Нет	Да	Да

Выводы

В результате анализа структуры дескриптора процесса выделены поля, связанные с планированием: *prio*, *rt_priority*, *normal_prio*, *static_prio*, *stats*, *sched_info*, *policy*, *se*, *rt*, *dl*, *sched_class*.

Проанализированы структуры *sched_statistics* и *sched_info*, содержащие статистическую информацию, связанную с планированием процесса.

Проанализированы структуры очередей процессов, участвующих в планировании: struct rq, struct cfs_rq, struct rt_rq, struct dl_rq, а также структуры, хранящие информацию о процессах в этих очередях: struct sched_entity, struct sched_rt_entity, struct sched_dl_entity. Также была выявлена связь между структурами очередей процессов.

Проведен анализ структуры *sched_class* и алгоритмов планирования: CFS, Deadline, Real-Time. Определены поля, влияющие на работу рассмотренных алгоритмов:

— для планировщика CFS — поле *vruntime*, находящееся в структуре *sched_entity*;

— для Deadline-планировщика — поля *dl_runtime*, *dl_deadline*, *dl_period*, *runtime*, *deadline*, хранящиеся в структуре *sched_dl_entity*;

— для Real-Time планировщика — поле *time_slice* из структуры *sched_rt_entity*.

В результате сравнительного анализа способов перехвата функций ядра выбраны интерфейс kprobes для функций, не возвращающих значение, и фреймворк ftrace для перехвата функций, возвращающих значение, так как они не требуют перекомпиляции ядра.

2 Конструкторский раздел

2.1 Последовательность действий

На рисунках 3 — 4 приведена последовательность действий при загрузке и выгрузке модуля.

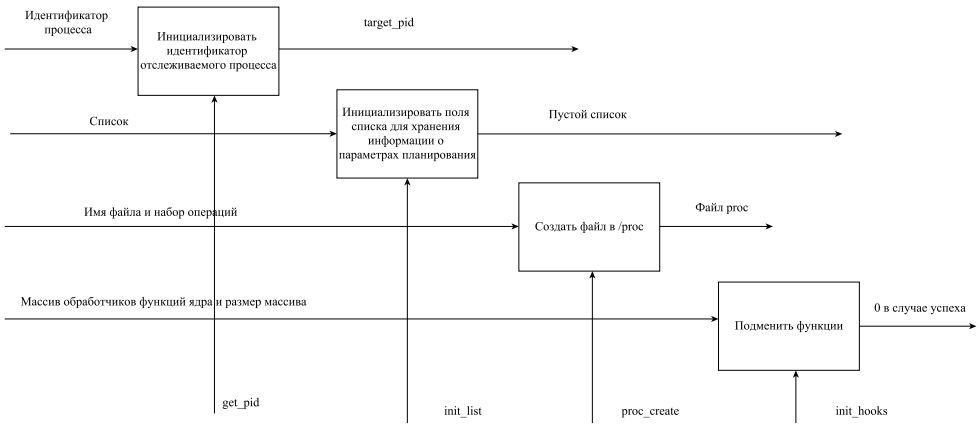

Рисунок 3 – Последовательность действий при загрузке модуля

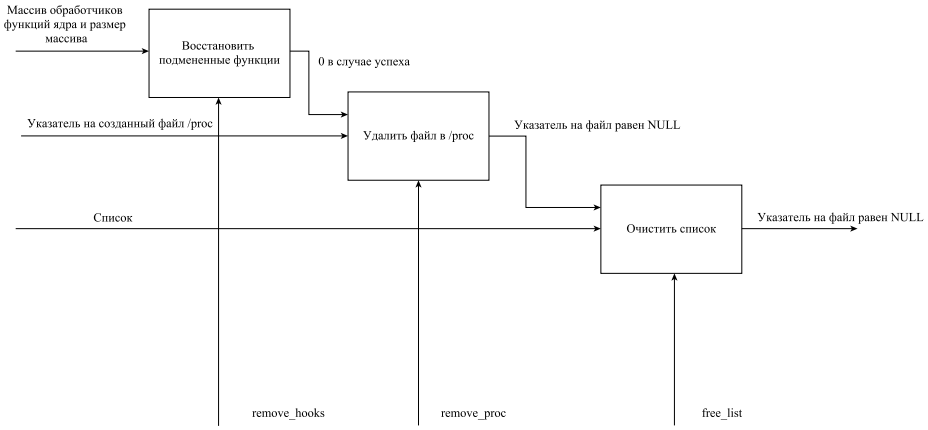

Рисунок 4 – Последовательность действий при выгрузке модуля

2.2 Разработка алгоритма работы предобработчика

На рисунке 5 приведена схема действий, выполняемых предобработчиком функции $task\_tick\_fair$.

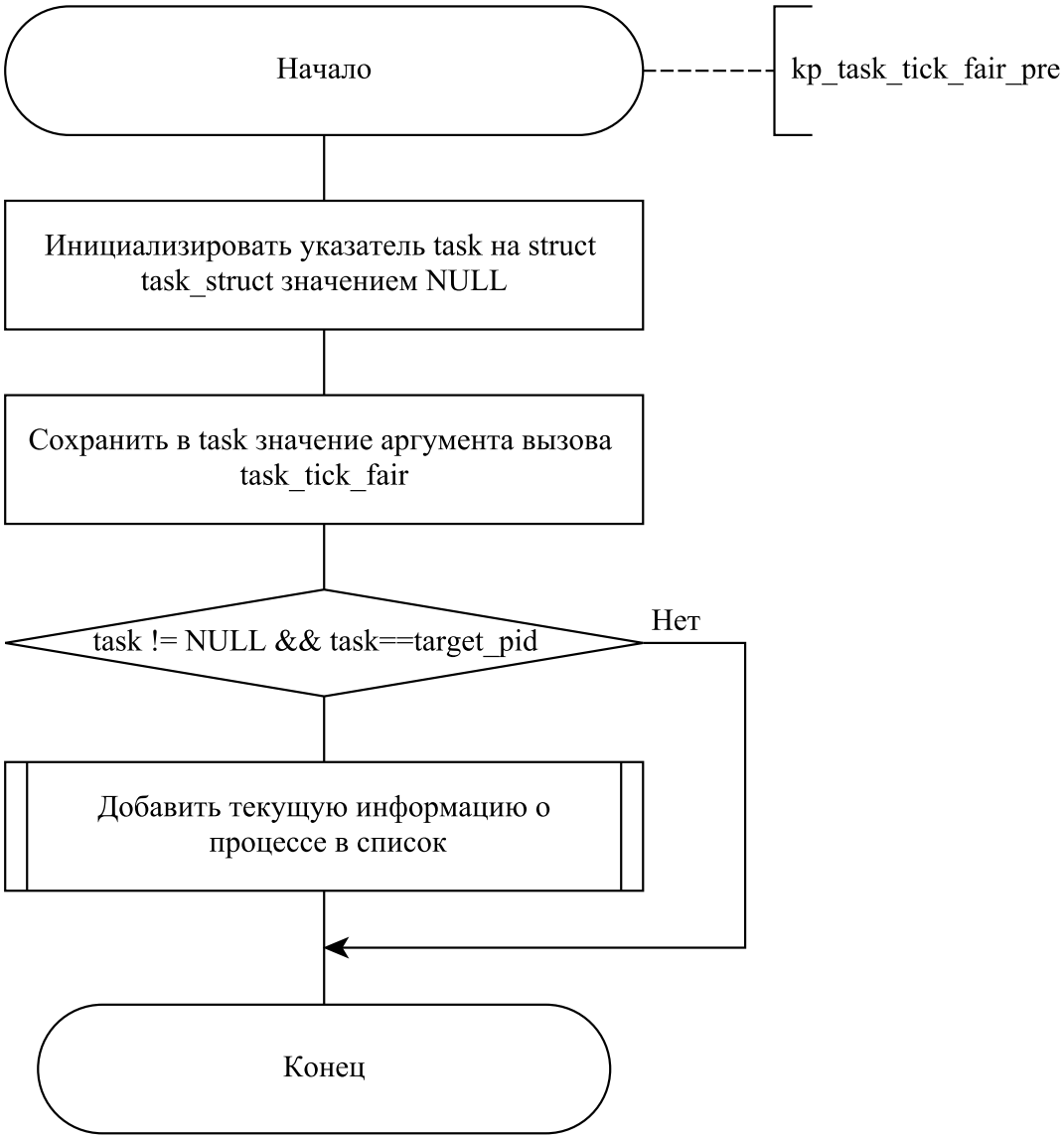

Рисунок 5 – Схема алгоритма работы предобработчика функции task_tick_fair

В предобработчике необходимо получить второй аргумент функции $task\_tick\_fair$, содержащий указатель на struct task_struct. Если указатель не NULL и поле *pid* совпадает с идентификатором искомого процесса, необходимо добавить информацию из структуры $task\_struct$ в список информации

о параметрах планирования.

2.3 Разработка алгоритма создания элемента списка

На рисунке 6 приведена схема создания элемента списка.

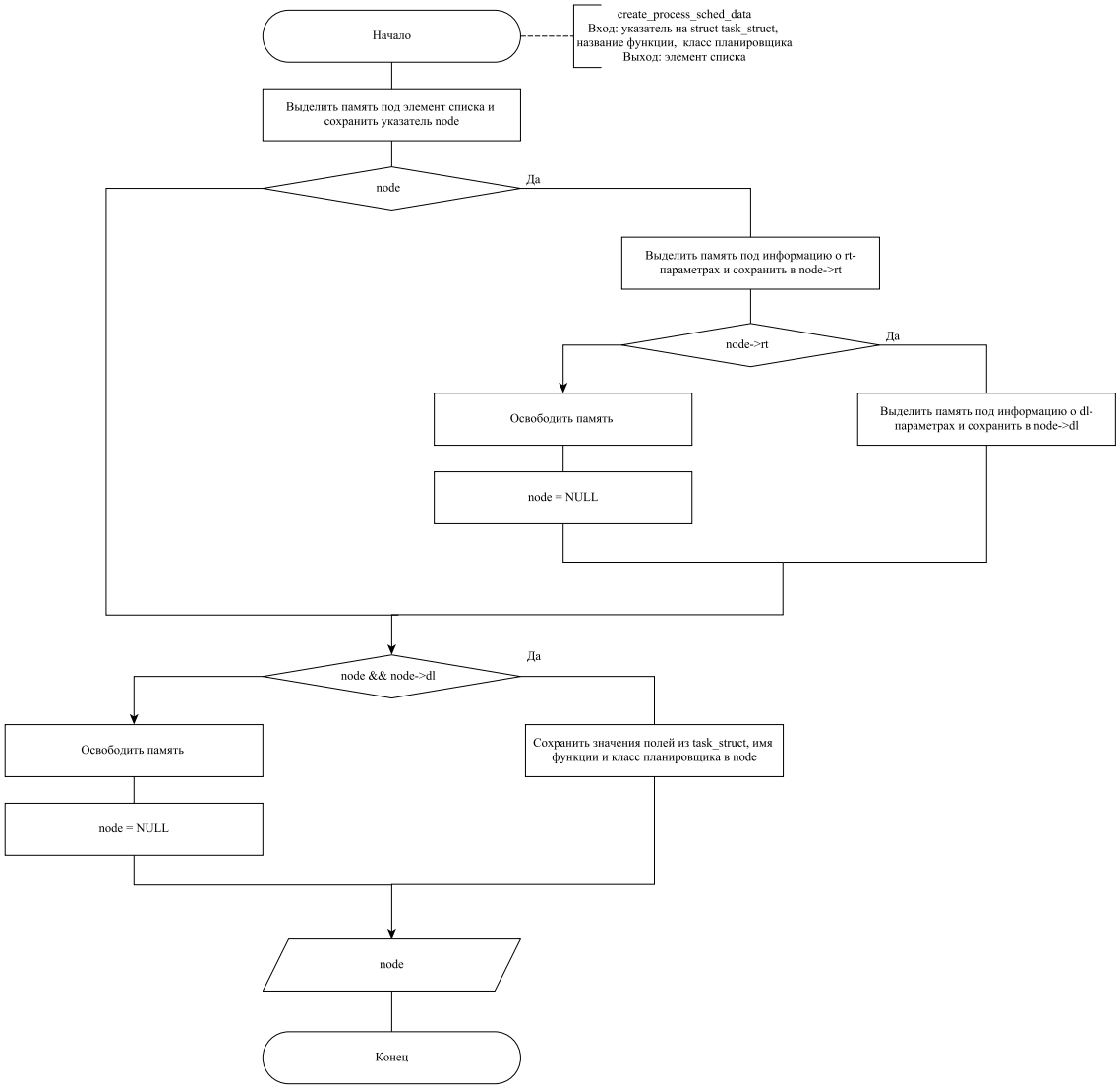

Рисунок 6 – Схема создания элемента списка

В случае успешного выделения памяти под элемент списка выделяется память под поля, хранящие информацию о параметрах планирования для Real-Time и Deadline планировщиков. Если при выделении памяти произошла ошибка, то память, выделенная под элемент списка, освобождается, а в

качестве возвращаемого значения используется NULL.

2.4 Структура программного обеспечения

Разрабатываемое ПО должно быть реализовано в виде загружаемого модуля ядра, в качестве параметра которого указывается идентификатор отслеживаемого процесса. Для передачи пользователю информации о планировании процесса докжен использоваться sequence-файл в *proc*. Структура программного обеспечения представлена на рисунке 7.

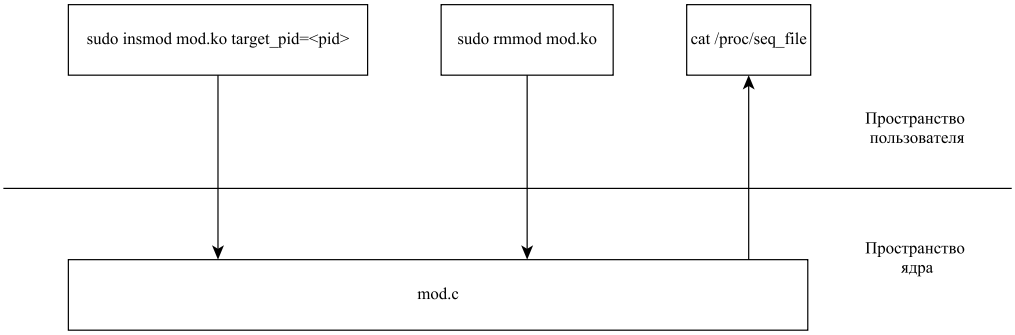

Рисунок 7 – Структура программного обеспечения

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для реализации был выбран язык C [7], так как в нем есть все средства для реализации загружаемого модуля ядра, выполняющего поставленную задачу. В качестве среды разработки был выбран Visual Studio Code [8], так как он обладает всеми необходимыми средствами для реализации программы на языке C.

3.2 Реализация структур для хранения информации о планировании

В листинге 16 представлены структуры для хранения информации о планировании процесса.

Листинг 16 – Структуры для хранения информации о планировании процесса

```
1  static int target_pid = 0;
2
3  #define MAX_FUNC_NAME_LENGTH 1024
4  #define MAX_SCHED_CLASS_LENGTH 3
5
6  #define RT_SCHED_CLASS 0
7  #define DL_SCHED_CLASS 1
8  #define CFS_SCHED_CLASS 2
9  #define IDLE_SCHED_CLASS 3
10
11 struct rt_sched_data {
12     unsigned long           timeout;
13     unsigned int            time_slice;
```

```
14 };
15
16 struct dl_sched_data {
17     u64 dl_runtime;
18     u64 dl_deadline;
19     u64 dl_period;
20     s64 runtime;
21     u64 deadline;
22 };
23
24 struct process_sched_data {
25     /* priorities from struct task_struct */
26     int prio;
27     int static_prio;
28     int normal_prio;
29     unsigned int rt_priority;
30
31     unsigned int policy;
32     /* weight from task->se.load.weight */
33     unsigned long weight;
34     /* task->se.vruntime */
35     u64 vruntime;
36
37     struct dl_sched_data *dl_data;
38     struct rt_sched_data *rt_data;
39     struct sched_info sched_info;
40     struct sched_statistics stats;
41     struct sched_entity se;
42     char func_name[MAX_FUNC_NAME_LENGTH];
43     int sched_class;
44
45     struct process_sched_data *next;
46 };
47
48 struct sched_data_list {
```

```
49      struct process_sched_data *head;
50      struct process_sched_data *tail;
51  };
52
53  struct sched_data_list sched_data_list;
```

Структуры *rt_sched_data* и *dl_sched_data* предназначены для хранения параметров Real-Time и Deadline планировщиков. Структура *process_sched_data* содержит информацию о параметрах планирования. Структура *sched_data_list* содержит указатели на первый и последний элементы списка.

3.3 Реализация создания элемента списка

В листинге 17 представлена функция создания элемента списка с информацией о планировании.

Листинг 17 – Функция создания элемента списка с информацией о планировании

```
1  struct process_sched_data * create_process_sched_data(struct
       task_struct *p, char *func_name, int sched_class)
2  {
3      struct process_sched_data *data = (struct process_sched_data
           *) kmalloc(sizeof(struct process_sched_data), GFP_KERNEL);
4
5      if (!data)
6          return NULL;
7
8      data->rt_data = NULL;
9      data->dl_data = NULL;
10
11     data->rt_data = (struct rt_sched_data *)
           kmalloc(sizeof(struct rt_sched_data), GFP_KERNEL);
```

```
12
13      if (!data->rt_data)
14      {
15          kfree(data);
16          return NULL;
17      }
18      data->rt_data->timeout = p->rt.timeout;
19      data->rt_data->time_slice = p->rt.time_slice;
20
21      data->dl_data = (struct dl_sched_data *)
           kmalloc(sizeof(struct dl_sched_data), GFP_KERNEL);
22
23      if (!data->dl_data)
24      {
25          kfree(data->rt_data);
26          kfree(data);
27          return NULL;
28      }
29
30      data->dl_data->dl_runtime = p->dl.dl_runtime;
31      data->dl_data->dl_deadline = p->dl.dl_deadline;
32      data->dl_data->dl_period = p->dl.dl_period;
33      data->dl_data->runtime = p->dl.runtime;
34      data->dl_data->deadline = p->dl.deadline;
35
36      data->next = NULL;
37      data->prio = p->prio;
38      data->static_prio = p->static_prio;
39      data->normal_prio = p->normal_prio;
40      data->rt_priority = p->rt_priority;
41      data->policy = p->policy;
42      data->sched_class = sched_class;
43      struct sched_statistics *stat =
           __my_schedstats_from_se(&p->se);
44      data->stats =  p->stats;
```

```
45      data->sched_info = p->sched_info;
46      data->se = p->se;
47      int read_len = snprintf(data->func_name,
            MAX_FUNC_NAME_LENGTH, "%s", &func_name[0]);
48
49      return data;
50  }
```

3.4 Реализация обработчика функции task_tick_fair

В листинге 18 представлен обработчик функции *task_tick_fair*.

Листинг 18 – Обработчик функции task_tick_fair

```
1  static int __kprobes kp_task_tick_fair_pre(struct kprobe *p,
       struct pt_regs *regs)
2  {
3      struct task_struct *task = NULL;
4      task = (struct task_struct *) p_regs_get_second_arg(regs);
5      if (task && task->pid == target_pid)
6      {
7          printk(KERN_INFO "before task_tick_fair() %d", task->pid);
8          add_sched_data(task, "task_tick_fair before",
               CFS_SCHED_CLASS);
9      }
10     return 0;
11 }
```

Функция *p_regs_get_second_arg* получает второй параметр, содержащий указатель на структуру *task_struct*, из регистров.

В приложении А приведен полный код программы.

4 Исследовательский раздел

Программное обеспечение реализовано на дистрибутиве KUBUNTU с ядром версии 6.8.0.

4.1 Демонстрация работы программы

Для получения результатов для процесса реального времени использовалось воспроизведение музыки в браузере Mozilla Firefox. На рисунке 8 приведена таблица параметров планирования для процесса реального времени.

function	policy	prio	static_prio	normal_prio	rt_priority	load_weight	vruntime	vlag	slice	rt_timeout	rt_time_slice	dl dl_runtime
enqueue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
pick_next_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
dequeue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
put_prev_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
enqueue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
pick_next_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
dequeue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
put_prev_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
enqueue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
pick_next_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
dequeue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
put_prev_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
enqueue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
pick_next_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
dequeue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
put_prev_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
update_curr_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
enqueue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
pick_next_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0
dequeue_task_rt	SCHED_RR	89	120	89	10	1048576	457350341693	-89156	1500000	0	49	0

Рисунок 8 – Демонстрация таблицы параметров планирования для процесса реального времени

Для получения результатов для процесса с дисциплиной планирования SCHED_NORMAL была использована программа, получающая на вход размеры двух матриц и их элементы и выводящая результат умножения. Ни рисунке 9 приведена таблица параметров планирования для процесса с дисциплиной планирования SCHED_NORMAL.

function	policy	prio	static_prio	normal_prio	rt_priority	load_weight	vruntime	vlag	slice	rt_timeout	rt_time_slice
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544390063	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544390063	2545890063	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544409996	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544409996	2545909996	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544411793	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544411793	2545911793	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544437039	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544437039	2545937039	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544441024	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544466960	2545941024	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544469979	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544469979	2545969979	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544471299	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	5195549098	5197030256	1500000	0	100
put_prev_task_fair	SCHED_NORMAL	120	120	120	0	1048576	5195553056	0	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	5195553056	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	5195553056	0	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	5195553056	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544471299	2545971299	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544625125	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544625125	2546125125	1500000	0	100
put_prev_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544626920	0	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544626920	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544626920	2546126920	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544645221	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544645221	2546145221	1500000	0	100
enqueue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544671864	0	1500000	0	100
dequeue_task_fair	SCHED_NORMAL	120	120	120	0	1048576	2544671864	2546171864	1500000	0	100

Рисунок 9 – Демонстрация таблицы параметров планирования для процесса с дисциплиной планирования SCHED_NORMAL

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы проведен анализа структуры дескриптора процесса и выделены поля, связанные с планированием: *prio*, *rt_priority*, *normal_prio*, *static_prio*, *stats*, *sched_info*, *policy*, *se*, *rt*, *dl*, *sched_class*. Проанализированы следующие структуры: *sched_statistics*, *sched_info*, *rq*, *cfs_rq*, *srt_rq*, *dl_rq*, *sched_entity*, *sched_rt_entity*, *sched_dl_entity*.

Проведен анализ структуры *sched_class* и алгоритмов планирования: CFS, Deadline, Real-Time. Определены поля, влияющие на работу рассмотренных алгоритмов:

— для планировщика CFS — поле *vruntime*, находящееся в структуре *sched_entity*;

— для Deadline планировщика — поля *dl_runtime*, *dl_deadline*, *dl_period*, *runtime*, *deadline*, хранящиеся в структуре *sched_dl_entity*;

— для Real-Time планировщика — поле *time_slice* из структуры *sched_rt_entity*.

Проведен сравнительный анализ способов перехвата функций ядра, в результате которого выбраны kprobes и ftrace, так как они не требуют перекомпиляции ядра и поддерживаются современными версиями ядра.

В рамках курсовой работы были выполнены поставленные задачи:

— проведен анализ структур и функций, предоставляющих возможность реализовать поставленную задачу;

— проведен анализ способов перехвата функций;

— разработаны алгоритмы и структура загружаемого модуля, обеспечивающего отслеживание работы планировщика;

— спроектирован и реализован загружаемый модуль ядра;

— проанализирована работа загружаемого модуля ядра.

Исследование разработанного программного обеспечения показало, что

оно соответствует техническому заданию и выполняет все поставленные задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux Process Scheduler - Basic [Электронный ресурс]. — Режим доступа: https://programming.vip/docs/original-linux-process-scheduler-basic.html (дата обращения: 08.02.2025).

2. Deadline Task Scheduling [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html (дата обращения: 05.02.2025).

3. Love R. Linux Kernel Development Second Edition. — Sams Publishing, 2005. — P. 432.

4. Linux kernel scheduler [Электронный ресурс]. — Режим доступа: https://helix979.github.io/jkoo/post/os-scheduler/ (дата обращения: 04.02.2025).

5. Перехват функций в ядре Linux с помощью ftrace [Электронный ресурс]. — Режим доступа: https://habr.com/ru/articles/413241/ (дата обращения: 02.02.2025).

6. Kernel Probes (Kprobes) [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/html/v6.6/trace/kprobes.html (дата обращения: 02.02.2025).

7. Стандарт C99 [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/c/99 (дата обращения: 04.02.2025).

8. Visual Studio Code [Электронный ресурс]. — Режим доступа: https://code.visualstudio.com/ (дата обращения: 04.02.2025).

ПРИЛОЖЕНИЕ А

Исходный код загружаемого модуля

Листинг 19 – Загружаемый модуль ядра

```
1
2 #include <linux/init.h>
3 #include <linux/sched.h>
4 #include <linux/sched/signal.h>
5 #include <linux/pid.h>
6 #include <linux/string.h>
7 #include <linux/types.h>
8 #include <linux/sched/types.h>
9 #include <linux/ftrace.h>
10 #include <linux/kallsyms.h>
11 #include <linux/kernel.h>
12 #include <linux/linkage.h>
13 #include <linux/module.h>
14 #include <linux/uaccess.h>
15 #include <linux/version.h>
16 #include <linux/kprobes.h>
17 #include <linux/proc_fs.h>
18 #include <linux/vmalloc.h>
19 #include <linux/seq_file.h>
20 #include <linux/fs_struct.h>
21
22 MODULE_LICENSE("GPL");
23 MODULE_AUTHOR("Vladimir Lebedev");
24 MODULE_DESCRIPTION("LKM for intercepting scheduler actions
    (Kernel 6.8.0)");
25
26 static int target_pid = 0;
27 module_param(target_pid, int, 0644);
28 MODULE_PARM_DESC(target_pid, "PID of the target process");
```

```
29
30 #define MAX_FUNC_NAME_LENGTH 1024
31 #define MAX_SCHED_CLASS_LENGTH 3
32 #define RT_SCHED_CLASS 0
33 #define DL_SCHED_CLASS 1
34 #define CFS_SCHED_CLASS 2
35 #define IDLE_SCHED_CLASS 3
36
37 struct my_uclamp_bucket {
38     unsigned long value : bits_per(SCHED_CAPACITY_SCALE);
39     unsigned long tasks : BITS_PER_LONG -
        bits_per(SCHED_CAPACITY_SCALE);
40 };
41
42 struct my_uclamp_rq {
43     unsigned int value;
44     struct my_uclamp_bucket bucket[UCLAMP_BUCKETS];
45 };
46
47 typedef int (*my_cpu_stop_fn_t)(void *arg);
48
49 struct my_cpu_stop_work {
50     struct list_head    list;
51     my_cpu_stop_fn_t        fn;
52     unsigned long       caller;
53     void            *arg;
54     struct cpu_stop_done    *done;
55 };
56
57 struct my_cfs_rq {
58     struct load_weight  load;
59     unsigned int        nr_running;
60     unsigned int        h_nr_running;
61     unsigned int        idle_nr_running;
62     unsigned int        idle_h_nr_running;
```

```
63      s64             avg_vruntime;
64      u64             avg_load;
65      u64             exec_clock;
66      u64             min_vruntime;
67      #ifdef CONFIG_SCHED_CORE
68      unsigned int            forceidle_seq;
69      u64             min_vruntime_fi;
70      #endif
71      #ifndef CONFIG_64BIT
72      u64             min_vruntime_copy;
73      #endif
74      struct rb_root_cached   tasks_timeline;
75      struct sched_entity *curr;
76      struct sched_entity *next;
77      #ifdef  CONFIG_SCHED_DEBUG
78      unsigned int            nr_spread_over;
79      #endif
80      #ifdef CONFIG_SMP
81
82      struct sched_avg    avg;
83      #ifndef CONFIG_64BIT
84      u64             last_update_time_copy;
85      #endif
86      struct {
87          raw_spinlock_t  lock ____cacheline_aligned;
88          int     nr;
89          unsigned long   load_avg;
90          unsigned long   util_avg;
91          unsigned long   runnable_avg;
92      } removed;
93      #ifdef CONFIG_FAIR_GROUP_SCHED
94      u64             last_update_tg_load_avg;
95      unsigned long           tg_load_avg_contrib;
96      long            propagate;
97      long            prop_runnable_sum;
```

```
98      unsigned long           h_load;
99      u64             last_h_load_update;
100     struct sched_entity *h_load_next;
101     #endif /* CONFIG_FAIR_GROUP_SCHED */
102     #endif /* CONFIG_SMP */
103
104     #ifdef CONFIG_FAIR_GROUP_SCHED
105     struct rq           *rq;
106     int             on_list;
107     struct list_head    leaf_cfs_rq_list;
108     struct task_group   *tg;
109
110     int             idle;
111
112     #ifdef CONFIG_CFS_BANDWIDTH
113     int             runtime_enabled;
114     s64             runtime_remaining;
115
116     u64             throttled_pelt_idle;
117     #ifndef CONFIG_64BIT
118     u64                         throttled_pelt_idle_copy;
119     #endif
120     u64             throttled_clock;
121     u64             throttled_clock_pelt;
122     u64             throttled_clock_pelt_time;
123     u64             throttled_clock_self;
124     u64             throttled_clock_self_time;
125     int             throttled;
126     int             throttle_count;
127     struct list_head    throttled_list;
128     struct list_head    throttled_csd_list;
129     #endif /* CONFIG_CFS_BANDWIDTH */
130     #endif /* CONFIG_FAIR_GROUP_SCHED */
131 };
132
```

```
133 struct my_rt_prio_array {
134     DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1);
135     struct list_head queue[MAX_RT_PRIO];
136 };
137 
138 struct my_rt_rq {
139     struct my_rt_prio_array active;
140     unsigned int        rt_nr_running;
141     unsigned int        rr_nr_running;
142     #if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
143     struct {
144         int     curr;
145         #ifdef CONFIG_SMP
146         int     next; /* next highest */
147         #endif
148     } highest_prio;
149     #endif
150     #ifdef CONFIG_SMP
151     int         overloaded;
152     struct plist_head   pushable_tasks;
153     #endif /* CONFIG_SMP */
154     int         rt_queued;
155     int         rt_throttled;
156     u64         rt_time;
157     u64         rt_runtime;
158     raw_spinlock_t      rt_runtime_lock;
159     #ifdef CONFIG_RT_GROUP_SCHED
160     unsigned int        rt_nr_boosted;
161     struct rq       *rq;
162     struct task_group   *tg;
163     #endif
164 };
165 
166 struct my_dl_rq {
167     struct rb_root_cached   root;
```

```
168     unsigned int            dl_nr_running;
169     #ifdef CONFIG_SMP
170     struct {
171         u64     curr;
172         u64     next;
173     } earliest_dl;
174     int             overloaded;
175     struct rb_root_cached   pushable_dl_tasks_root;
176     #else
177     struct dl_bw            dl_bw;
178     #endif
179     u64             running_bw;
180     u64             this_bw;
181     u64             extra_bw;
182     u64             max_bw;
183     u64             bw_ratio;
184 };
185
186 struct my_rq {
187     raw_spinlock_t          __lock;
188
189     unsigned int            nr_running;
190     #ifdef CONFIG_NUMA_BALANCING
191     unsigned int            nr_numa_running;
192     unsigned int            nr_preferred_running;
193     unsigned int            numa_migrate_on;
194     #endif
195     #ifdef CONFIG_NO_HZ_COMMON
196     #ifdef CONFIG_SMP
197     unsigned long           last_blocked_load_update_tick;
198     unsigned int            has_blocked_load;
199     call_single_data_t      nohz_csd;
200     #endif /* CONFIG_SMP */
201     unsigned int            nohz_tick_stopped;
202     atomic_t            nohz_flags;
```

```
203     #endif /* CONFIG_NO_HZ_COMMON */
204
205     #ifdef CONFIG_SMP
206     unsigned int            ttwu_pending;
207     #endif
208     u64             nr_switches;
209
210     #ifdef CONFIG_UCLAMP_TASK
211     struct my_uclamp_rq uclamp[UCLAMP_CNT] _____cacheline_aligned;
212     unsigned int            uclamp_flags;
213     #define UCLAMP_FLAG_IDLE 0x01
214     #endif
215
216     struct my_cfs_rq            cfs;
217     struct my_rt_rq         rt;
218     struct my_dl_rq         dl;
219
220     #ifdef CONFIG_FAIR_GROUP_SCHED
221     /* list of leaf cfs_rq on this CPU: */
222     struct list_head        leaf_cfs_rq_list;
223     struct list_head        *tmp_alone_branch;
224     #endif /* CONFIG_FAIR_GROUP_SCHED */
225     unsigned int            nr_uninterruptible;
226
227     struct task_struct __rcu    *curr;
228     struct task_struct  *idle;
229     struct task_struct  *stop;
230     unsigned long           next_balance;
231     struct mm_struct        *prev_mm;
232
233     unsigned int            clock_update_flags;
234     u64             clock;
235     u64             clock_task _____cacheline_aligned;
236     u64             clock_pelt;
237     unsigned long           lost_idle_time;
```

```
238     u64             clock_pelt_idle;
239     u64             clock_idle;
240     #ifndef CONFIG_64BIT
241     u64             clock_pelt_idle_copy;
242     u64             clock_idle_copy;
243     #endif
244     atomic_t        nr_iowait;
245     #ifdef CONFIG_SCHED_DEBUG
246     u64 last_seen_need_resched_ns;
247     int ticks_without_resched;
248     #endif
249     #ifdef CONFIG_MEMBARRIER
250     int membarrier_state;
251     #endif
252     #ifdef CONFIG_SMP
253     struct root_domain      *rd;
254     struct sched_domain __rcu   *sd;
255     unsigned long       cpu_capacity;
256     struct balance_callback *balance_callback;
257     unsigned char       nohz_idle_balance;
258     unsigned char       idle_balance;
259     unsigned long       misfit_task_load;
260     int             active_balance;
261     int             push_cpu;
262     struct my_cpu_stop_work active_balance_work;
263     int             cpu;
264     int             online;
265     struct list_head cfs_tasks;
266     struct sched_avg    avg_rt;
267     struct sched_avg    avg_dl;
268     #ifdef CONFIG_HAVE_SCHED_AVG_IRQ
269     struct sched_avg    avg_irq;
270     #endif
271     #ifdef CONFIG_SCHED_THERMAL_PRESSURE
272     struct sched_avg    avg_thermal;
```

```
273     #endif
274     u64             idle_stamp;
275     u64             avg_idle;
276     u64             max_idle_balance_cost;
277     #ifdef CONFIG_HOTPLUG_CPU
278     struct rcuwait      hotplug_wait;
279     #endif
280     #endif /* CONFIG_SMP */
281     #ifdef CONFIG_IRQ_TIME_ACCOUNTING
282     u64             prev_irq_time;
283     #endif
284     #ifdef CONFIG_PARAVIRT
285     u64             prev_steal_time;
286     #endif
287     #ifdef CONFIG_PARAVIRT_TIME_ACCOUNTING
288     u64             prev_steal_time_rq;
289     #endif
290     unsigned long       calc_load_update;
291     long            calc_load_active;
292     #ifdef CONFIG_SCHED_HRTICK
293     #ifdef CONFIG_SMP
294     call_single_data_t  hrtick_csd;
295     #endif
296     struct hrtimer      hrtick_timer;
297     ktime_t         hrtick_time;
298     #endif
299     #ifdef CONFIG_SCHEDSTATS
300     /* latency stats */
301     struct sched_info   rq_sched_info;
302     unsigned long long  rq_cpu_time;
303     unsigned int        yld_count;
304     unsigned int        sched_count;
305     unsigned int        sched_goidle;
306     unsigned int        ttwu_count;
307     unsigned int        ttwu_local;
```

```
308     #endif
309     #ifdef CONFIG_CPU_IDLE
310     struct cpuidle_state    *idle_state;
311     #endif
312     #ifdef CONFIG_SMP
313     unsigned int            nr_pinned;
314     #endif
315     unsigned int            push_busy;
316     struct my_cpu_stop_work push_work;
317     #ifdef CONFIG_SCHED_CORE
318     struct rq       *core;
319     struct task_struct  *core_pick;
320     unsigned int            core_enabled;
321     unsigned int            core_sched_seq;
322     struct rb_root      core_tree;
323     unsigned int            core_task_seq;
324     unsigned int            core_pick_seq;
325     unsigned long       core_cookie;
326     unsigned int            core_forceidle_count;
327     unsigned int            core_forceidle_seq;
328     unsigned int            core_forceidle_occupation;
329     u64         core_forceidle_start;
330     #endif
331     cpumask_var_t       scratch_mask;
332     #if defined(CONFIG_CFS_BANDWIDTH) && defined(CONFIG_SMP)
333     call_single_data_t  cfsb_csd;
334     struct list_head    cfsb_csd_list;
335     #endif
336 };
337
338 struct rt_sched_data {
339     unsigned long           timeout;
340     unsigned int            time_slice;
341 };
342
```

```
343 struct dl_sched_data {
344     u64 dl_runtime;
345     u64 dl_deadline;
346     u64 dl_period;
347     s64 runtime;
348     u64 deadline;
349 };
350
351 struct process_sched_data {
352     int prio;
353     int static_prio;
354     int normal_prio;
355     unsigned int rt_priority;
356     unsigned int policy;
357     unsigned long weight;
358     u64 vruntime;
359     struct dl_sched_data *dl_data;
360     struct rt_sched_data *rt_data;
361     struct sched_info sched_info;
362     struct sched_statistics stats;
363     struct sched_entity se;
364     char func_name[MAX_FUNC_NAME_LENGTH];
365     int sched_class;
366     struct process_sched_data *next;
367 };
368
369 struct sched_data_list {
370     struct process_sched_data *head;
371     struct process_sched_data *tail;
372 };
373
374 struct sched_data_list sched_data_list;
375
376 struct ftrace_hook {
377     const char *name;
```

```
378     void *function;
379     void *original;
380 
381     unsigned long address;
382     struct ftrace_ops ops;
383 };
384 
385 void print_task_info(struct task_struct *task);
386 int fh_install_hook(struct ftrace_hook *hook);
387 void fh_remove_hook(struct ftrace_hook *hook);
388 int fh_install_hooks(struct ftrace_hook *hooks, size_t count);
389 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count);
390 int install_hooks(void);
391 void remove_hooks(void);
392 struct process_sched_data * create_process_sched_data(struct
       task_struct *p, char *func_name, int sched_class);
393 int push_data(struct process_sched_data *data);
394 int add_sched_data(struct task_struct *p, char *func_name, int
       sched_class);
395 void pop_data(void);
396 void free_process_sched_data_list(void);
397 void free_process_sched_data(struct process_sched_data *data);
398 void init_process_sched_data_list(void);
399 unsigned long p_regs_get_first_arg(struct pt_regs* regs);
400 unsigned long p_regs_get_second_arg(struct pt_regs* regs);
401 unsigned long p_regs_get_third_arg(struct pt_regs* regs);
402 
403 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
404 static unsigned long lookup_name(const char *name)
405 {
406     struct kprobe kp = {
407         .symbol_name = name
408     };
409     unsigned long retval;
410 
```

```
411     if (register_kprobe(&kp) < 0) return 0;
412     retval = (unsigned long) kp.addr;
413     unregister_kprobe(&kp);
414     return retval;
415 }
416 #else
417 static unsigned long lookup_name(const char *name)
418 {
419     return kallsyms_lookup_name(name);
420 }
421 #endif
422
423 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
424 #define FTRACE_OPS_FL_RECURSION FTRACE_OPS_FL_RECURSION_SAFE
425 #endif
426
427 #if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >=
       KERNEL_VERSION(4,17,0))
428 #define PTREGS_SYSCALL_STUBS 1
429 #endif
430
431 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
432 #define ftrace_regs pt_regs
433
434 static __always_inline struct pt_regs *ftrace_get_regs(struct
       ftrace_regs *fregs)
435 {
436     return fregs;
437 }
438 #endif
439
440 #define USE_FENTRY_OFFSET 0
441
442 static int fh_resolve_hook_address(struct ftrace_hook *hook)
443 {
```

```
444     hook->address = lookup_name(hook->name);
445     if (!hook->address) {
446         printk(KERN_INFO "-> unresolved symbol: %s\n",
                hook->name);
447         return -ENOENT;
448     }
449     #if USE_FENTRY_OFFSET
450     *((unsigned long*) hook->original) = hook->address +
            MCOUNT_INSN_SIZE;
451     #else
452     *((unsigned long*) hook->original) = hook->address;
453     #endif
454     return 0;
455 }
456
457 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned
        long parent_ip,
458 struct ftrace_ops *ops, struct ftrace_regs *fregs)
459 {
460     struct pt_regs *regs = ftrace_get_regs(fregs);
461     struct ftrace_hook *hook = container_of(ops, struct
            ftrace_hook, ops);
462     #if USE_FENTRY_OFFSET
463     regs->ip = (unsigned long)hook->function;
464     #else
465     if (!within_module(parent_ip, THIS_MODULE))
466     regs->ip = (unsigned long)hook->function;
467     #endif
468 }
469
470 int fh_install_hook(struct ftrace_hook *hook)
471 {
472     int err;
473     err = fh_resolve_hook_address(hook);
474     if (err)
```

```
475         return err;
476     hook->ops.func = fh_ftrace_thunk;
477     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
478     | FTRACE_OPS_FL_RECURSION
479     | FTRACE_OPS_FL_IPMODIFY;
480     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
481     if (err) {
482         printk(KERN_INFO "-> ftrace_set_filter_ip() failed:
              %d\n", err);
483         return err;
484     }
485     err = register_ftrace_function(&hook->ops);
486     if (err) {
487         printk(KERN_INFO "-> register_ftrace_function() failed:
              %d\n", err);
488         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
489         return err;
490     }
491     return 0;
492 }
493
494 void fh_remove_hook(struct ftrace_hook *hook)
495 {
496     int err;
497     err = unregister_ftrace_function(&hook->ops);
498     if (err) {
499         printk(KERN_INFO "-> unregister_ftrace_function() failed:
              %d\n", err);
500     }
501     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
502     if (err) {
503         printk(KERN_INFO "-> ftrace_set_filter_ip() failed:
              %d\n", err);
504     }
505 }
```

```
506
507 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
508 {
509     int err;
510     size_t i;
511     for (i = 0; i < count; i++) {
512         err = fh_install_hook(&hooks[i]);
513         if (err)
514         goto error;
515     }
516     return 0;
517 error:
518     while (i != 0) {
519         fh_remove_hook(&hooks[--i]);
520     }
521     return err;
522 }
523
524 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
525 {
526     size_t i;
527
528     for (i = 0; i < count; i++)
529     fh_remove_hook(&hooks[i]);
530 }
531
532 unsigned long p_regs_get_first_arg(struct pt_regs* regs)
533 {
534     return regs->di;
535 }
536
537 unsigned long p_regs_get_second_arg(struct pt_regs* regs)
538 {
539     return regs->si;
540 }
```

```
541
542 unsigned long p_regs_get_third_arg(struct pt_regs* regs)
543 {
544     return regs->dx;
545 }
546
547 #if !USE_FENTRY_OFFSET
548 #pragma GCC optimize("-fno-optimize-sibling-calls")
549 #endif
550
551 #ifndef CONFIG_X86_64
552 #error Currently only x86_64 architecture is supported
553 #endif
554
555 #ifdef PTREGS_SYSCALL_STUBS
556 #define SYSCALL_NAME(name) ("__x64_" name)
557 #else
558 #define SYSCALL_NAME(name) (name)
559 #endif
560
561 #define task_of(_se)    container_of(_se, struct task_struct, se)
562
563 static struct sched_statistics *
564 __my_schedstats_from_se(struct sched_entity *se)
565 {
566     return &task_of(se)->stats;
567 }
568
569 struct process_sched_data * create_process_sched_data(struct
       task_struct *p, char *func_name, int sched_class)
570 {
571     struct process_sched_data *data = (struct process_sched_data
          *) kmalloc(sizeof(struct process_sched_data), GFP_KERNEL);
572     if (!data)
573     return NULL;
```

```
574     data->rt_data = NULL;
575     data->dl_data = NULL;
576     data->rt_data = (struct rt_sched_data *)
           kmalloc(sizeof(struct rt_sched_data), GFP_KERNEL);
577     if (!data->rt_data)
578     {
579         kfree(data);
580         return NULL;
581     }
582     data->rt_data->timeout = p->rt.timeout;
583     data->rt_data->time_slice = p->rt.time_slice;
584     data->dl_data = (struct dl_sched_data *)
           kmalloc(sizeof(struct dl_sched_data), GFP_KERNEL);
585     if (!data->dl_data)
586     {
587         kfree(data->rt_data);
588         kfree(data);
589         return NULL;
590     }
591     data->dl_data->dl_runtime = p->dl.dl_runtime;
592     data->dl_data->dl_deadline = p->dl.dl_deadline;
593     data->dl_data->dl_period = p->dl.dl_period;
594     data->dl_data->runtime = p->dl.runtime;
595     data->dl_data->deadline = p->dl.deadline;
596     data->next = NULL;
597     data->prio = p->prio;
598     data->static_prio = p->static_prio;
599     data->normal_prio = p->normal_prio;
600     data->rt_priority = p->rt_priority;
601     data->policy = p->policy;
602     data->sched_class = sched_class;
603     struct sched_statistics *stat =
           __my_schedstats_from_se(&p->se);
604     data->stats =  p->stats;
605     data->sched_info = p->sched_info;
```

```
606     data->se = p->se;
607     int read_len = snprintf(data->func_name,
            MAX_FUNC_NAME_LENGTH, "%s", &func_name[0]);
608     return data;
609 }
610
611 void init_process_sched_data_list(void)
612 {
613     sched_data_list.head = NULL;
614     sched_data_list.tail = NULL;
615 }
616
617 int push_data(struct process_sched_data *data)
618 {
619     if (!data)
620     return -1;
621     if (!sched_data_list.head)
622     {
623         sched_data_list.head = data;
624         sched_data_list.tail = data;
625         return 0;
626     }
627
628     sched_data_list.tail->next = data;
629     sched_data_list.tail = data;
630
631     return 0;
632 }
633
634 int add_sched_data(struct task_struct *p, char *func_name, int
        sched_class)
635 {
636     struct process_sched_data *node =
            create_process_sched_data(p, func_name, sched_class);
637     if (!node)
```

```
638     {
639         return -1;
640     }
641     return push_data(node);
642 }
643
644 void pop_data(void)
645 {
646     if (!sched_data_list.head)
647     {
648         return;
649     }
650     struct process_sched_data *prev = sched_data_list.head;
651
652     while (prev->next != sched_data_list.tail)
653     {
654         prev = prev->next;
655     }
656     free_process_sched_data(sched_data_list.tail);
657     if (prev != sched_data_list.head)
658     {
659         prev->next = NULL;
660         sched_data_list.tail = prev;
661     }
662     else
663     {
664         sched_data_list.head = NULL;
665         sched_data_list.tail = NULL;
666     }
667 }
668
669 void free_process_sched_data_list(void)
670 {
671     while (sched_data_list.head)
672     {
```

```
673             pop_data();
674         }
675 }
676
677 void free_process_sched_data(struct process_sched_data *data)
678 {
679         if (!data)
680         return;
681         kfree(data->dl_data);
682         kfree(data->rt_data);
683         kfree(data);
684 }
685
686 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,6,0)
687 #define HAVE_PROC_OPS
688 #endif
689
690 #define FILENAME "seq_file"
691
692 static struct proc_dir_entry *seq_file;
693
694 static char *get_policy(int policy)
695 {
696         switch (policy)
697         {
698             case SCHED_NORMAL:
699             return "SCHED_NORMAL";
700             case SCHED_FIFO:
701             return "SCHED_FIFO";
702             case SCHED_RR:
703             return "SCHED_RR";
704             case SCHED_BATCH:
705             return "SCHED_BATCH";
706             case SCHED_IDLE:
707             return "SCHED_IDLE";
```

```
        case SCHED_DEADLINE:
        return "SCHED_DEADLINE";
        default:
        return "UNKNOWN";
        break;
    }
}

u64 get_avg(u64 sum, u64 count)
{
    if (count == 0)
    return 0;
    return sum / count;
}

static int seq_show(struct seq_file *m, void *v)
{
    printk(KERN_INFO "+seq: show\n");
    seq_printf(m, "|            function            | policy      | prio | static_prio | normal_prio | rt_priority | load_weight |    vruntime    "
    "|      vlag      |      slice      | rt_timeout | rt_time_slice | dl dl_runtime | dl dl_deadline | dl dl_period | dl runtime | dl deadline |"
    "    p_count     |    run_delay    |    last_arrival    | last_queued |    wait_max    |    wait_avg    | iowait_avg    "
    "|    sleep_max    |    sleep_sum    |    block_max    | block_sum    |    exec_max    |    slice_max    |\n");
    seq_printf(m,
        "----------------------------------------------------------------------------------------------------
    "----------------------------------------------------------------------------------------------------
    "----------------------------------------------------------------------------------------------------
    "----------------------------------------------------------------------------------------------------
    if (!sched_data_list.head)
```

```
735     return 0;
736     struct process_sched_data *curr = sched_data_list.head;
737     while (curr)
738     {
739         seq_printf(m, "| %33s | %14s | %5d | %11d | %11d | %11u |
               %11ld | %14lld | %14lld | %14lld | %10ld | %13u |
               %13lld | %14lld | %12lld | %10lld | %11lld | %13ld |
               %13lld | %16lld | %13lld | %14lld | %14lld | %14lld |
               %14lld | %14lld | %14lld | %14lld | %14lld | %14lld
               |\n",
740         curr->func_name, get_policy(curr->policy), curr->prio,
               curr->static_prio, curr->normal_prio,
               curr->rt_priority,
741         curr->se.load.weight, curr->se.vruntime, curr->se.vlag,
               curr->se.slice, curr->rt_data->timeout,
               curr->rt_data->time_slice,
742         curr->dl_data->dl_runtime, curr->dl_data->dl_deadline,
               curr->dl_data->dl_period, curr->dl_data->runtime,
               curr->dl_data->deadline,
743         curr->sched_info.pcount, curr->sched_info.run_delay,
               curr->sched_info.last_arrival,
744         curr->sched_info.last_queued, curr->stats.wait_max,
               get_avg(curr->stats.wait_sum, curr->stats.wait_count),
745         get_avg(curr->stats.iowait_sum,
               curr->stats.iowait_count), curr->stats.sleep_max,
               curr->stats.sum_sleep_runtime,
746         curr->stats.block_max, curr->stats.sum_block_runtime,
               curr->stats.exec_max, curr->stats.slice_max
747         );
748         curr = curr->next;
749     }
750
751     return 0;
752 }
753
```

```
754 static int seq_file_open(struct inode *inode, struct file *file)
755 {
756     printk(KERN_INFO "+seq: open\n");
757     return single_open(file, seq_show, NULL);
758 }
759
760 static ssize_t seq_file_write(struct file *file, const char
      __user *buf, size_t len, loff_t *fpos)
761 {
762     return len;
763 }
764
765 #ifdef HAVE_PROC_OPS
766 static const struct proc_ops fops =
767 {
768     .proc_open = seq_file_open,
769     .proc_release = single_release,
770     .proc_write = seq_file_write,
771     .proc_read = seq_read
772 };
773 #else
774 static const struct file_operations fops =
775 {
776     .open = seq_file_open,
777     .release = single_release,
778     .write = seq_file_write,
779     .read = seq_read
780 };
781 #endif
782
783 #ifdef PTREGS_SYSCALL_STUBS
784 static bool (*real_yield_to_task_fair)(struct pt_regs *regs);
785
786 static bool fh_yield_to_task_fair(struct pt_regs *regs)
787 {
```

```
788     bool ret = false;
789     struct task_struct *p = NULL;
790     p = (struct task_struct *) p_regs_get_second_arg(regs);
791     ret = real_yield_to_task_fair(regs);
792     if (ret && p && p->pid == target_pid)
793     {
794         printk(KERN_INFO "yield_to_task_fair stub() %d", p->pid);
795         add_sched_data(p, "yield_to_task_fair", CFS_SCHED_CLASS);
796     }
797     return ret;
798 }
799
800 static struct task_struct * (*real_pick_next_task_fair)(struct
       pt_regs *regs);
801 static struct task_struct * (*real_pick_next_task_rt)(struct
       pt_regs *regs);
802 static struct task_struct * (*real_pick_next_task_idle)(struct
       pt_regs *regs);
803 static struct task_struct * (*real_pick_next_task_dl)(struct
       pt_regs *regs);
804
805 static struct task_struct * fh_pick_next_task_fair(struct pt_regs
       *regs)
806 {
807     struct task_struct *p = NULL;
808     p = (struct task_struct *) real_pick_next_task_fair(regs);
809     if (p && p->pid == target_pid)
810     {
811         add_sched_data(p, "pick_next_task_fair", CFS_SCHED_CLASS);
812     }
813     return p;
814 }
815
816 static struct task_struct * fh_pick_next_task_rt(struct pt_regs
       *regs)
```

```
817 {
818     struct task_struct *p = NULL;
819     p = (struct task_struct *) real_pick_next_task_rt(regs);
820     if (p && p->pid == target_pid)
821     {
822         add_sched_data(p, "pick_next_task_rt", RT_SCHED_CLASS);
823     }
824     return p;
825 }
826
827 static struct task_struct * fh_pick_next_task_dl(struct pt_regs
       *regs)
828 {
829     struct task_struct *p = NULL;
830     p = (struct task_struct *) real_pick_next_task_dl(regs);
831     if (p && p->pid == target_pid)
832     {
833         add_sched_data(p, "pick_next_task_dl", DL_SCHED_CLASS);
834     }
835     return p;
836 }
837
838 static struct task_struct * fh_pick_next_task_idle(struct pt_regs
       *regs)
839 {
840     struct task_struct *p = NULL;
841     p = (struct task_struct *) real_pick_next_task_idle(regs);
842     if (p && p->pid == target_pid)
843     {
844         add_sched_data(p, "pick_next_task_idle",
                IDLE_SCHED_CLASS);
845     }
846     return p;
847 }
848 #else
```

```
849 static bool (*real_yield_to_task_fair) (struct rq *rq, struct
       task_struct *p);
850
851 static bool yield_to_task_fair(struct rq *rq, struct task_struct
       *p)
852 {
853     printk(KERN_INFO "yield_to_task_fair()");
854     bool ret = real_yield_to_task_fair(rq, p);
855     return ret;
856 }
857
858 static struct task_struct * (*real_pick_next_task_fair) (struct
       rq *rq);
859 static struct task_struct * (*real_pick_next_task_rt) (struct rq
       *rq);
860 static struct task_struct * (*real_pick_next_task_dl) (struct rq
       *rq);
861 static struct task_struct * (*real_pick_next_task_idle) (struct
       rq *rq);
862
863 static struct task_struct * fh_pick_next_task_fair(struct rq *rq)
864 {
865     printk(KERN_INFO "__pick_next_task_fair()");
866     return real_pick_next_task_fair(rq);
867 }
868
869 static struct task_struct * fh_pick_next_task_rt(struct rq *rq)
870 {
871     printk(KERN_INFO "pick_next_task_rt()");
872     return real_pick_next_task_rt(rq);
873 }
874
875 static struct task_struct * fh_pick_next_task_dl(struct rq *rq)
876 {
877     printk(KERN_INFO "pick_next_task_dl()");
```

```
878     return real_pick_next_task_dl(rq);
879 }
880
881 static struct task_struct * fh_pick_next_task_idle(struct rq *rq)
882 {
883     printk(KERN_INFO "pick_next_task_idle()");
884     return real_pick_next_task_idle(rq);
885 }
886 #endif
887
888 static void __kprobes kp_check_preempt_fair_post(struct kprobe
       *p, struct pt_regs *regs,
889 unsigned long flags)
890 {
891     struct task_struct *curr = NULL;
892     struct task_struct *task = NULL;
893     struct my_rq *rq = NULL;
894     rq = (struct my_rq *) regs->di;
895     if (!rq)
896     return;
897     curr = (struct task_struct *) rq->curr;
898     if (!curr)
899     return;
900     task = (struct task_struct *) regs->si;
901     if (!task)
902     return;
903     if (test_tsk_need_resched(curr) && task->pid == target_pid)
904     {
905         printk(KERN_INFO "check_preempt_wakeup_fair, pid %d, task
               pid %d", curr->pid, task->pid);
906         add_sched_data(task, "check_preempt_wakeup_fair preempt",
               CFS_SCHED_CLASS);
907     }
908     if (test_tsk_need_resched(curr) && curr->pid == target_pid)
909     {
```

```
910         printk(KERN_INFO "check_preempt_wakeup_fair, pid %d, task
                pid %d", curr->pid, task->pid);
911         add_sched_data(task, "check_preempt_wakeup_fair curr",
                CFS_SCHED_CLASS);
912     }
913 }
914
915 static int __kprobes kp_wakeup_preempt_dl_pre(struct kprobe *p,
        struct pt_regs *regs)
916 {
917     struct task_struct *curr = NULL;
918     struct task_struct *task = NULL;
919     struct my_rq *rq = NULL;
920     rq = (struct my_rq *) regs->di;
921     if (!rq)
922     return 0;
923     curr = (struct task_struct *) rq->curr;
924     if (!curr)
925     return 0;
926     task = (struct task_struct *) regs->si;
927     if (!task)
928     return 0;
929     if (task->pid == target_pid)
930     {
931         printk(KERN_INFO "wakeup_preempt_dl, pid %d, task pid
                %d", curr->pid, task->pid);
932         add_sched_data(task, "wakeup_preempt_dl preempt",
                DL_SCHED_CLASS);
933     }
934     else if (curr->pid == target_pid)
935     {
936         printk(KERN_INFO "wakeup_preempt_dl, pid %d, task pid
                %d", curr->pid, task->pid);
937         add_sched_data(task, "wakeup_preempt_dl curr",
                DL_SCHED_CLASS);
```

```
938     }
939     return 0;
940 }
941
942 static int __kprobes kp_wakeup_preempt_idle_pre(struct kprobe *p,
      struct pt_regs *regs)
943 {
944     struct task_struct *curr = NULL;
945     struct task_struct *task = NULL;
946     struct my_rq *rq = NULL;
947     rq = (struct my_rq *) regs->di;
948     if (!rq)
949     return 0;
950     curr = (struct task_struct *) rq->curr;
951     if (!curr)
952     return 0;
953     task = (struct task_struct *) regs->si;
954     if (!task)
955     return 0;
956     if (task->pid == target_pid)
957     {
958         printk(KERN_INFO "wakeup_preempt_idle, pid %d, task pid
              %d", curr->pid, task->pid);
959         add_sched_data(task, "check_preempt_idle preempt",
              IDLE_SCHED_CLASS);
960     }
961     else if (curr->pid == target_pid)
962     {
963         printk(KERN_INFO "wakeup_preempt_idle, pid %d, task pid
              %d", curr->pid, task->pid);
964         add_sched_data(task, "wakeup_preempt_idle curr",
              IDLE_SCHED_CLASS);
965     }
966     return 0;
967 }
```

```
968
969 static int __kprobes kp_wakeup_preempt_rt_pre(struct kprobe *p,
       struct pt_regs *regs)
970 {
971     struct task_struct *curr = NULL;
972     struct task_struct *task = NULL;
973     struct my_rq *rq = NULL;
974     rq = (struct my_rq *) p_regs_get_first_arg(regs);
975     if (!rq)
976     return 0;
977     curr = (struct task_struct *) rq->curr;
978     if (!curr)
979     return 0;
980     task = (struct task_struct *) p_regs_get_second_arg(regs);
981     if (!task)
982     return 0;
983     if (task->pid == target_pid)
984     {
985         printk(KERN_INFO "wakeup_preempt_rt, pid %d, task pid
              %d", curr->pid, task->pid);
986         add_sched_data(task, "wakeup_preempt_rt preempt",
              RT_SCHED_CLASS);
987     }
988     else if (curr->pid == target_pid)
989     {
990         printk(KERN_INFO "wakeup_preempt_rt, pid %d, task pid
              %d", curr->pid, task->pid);
991         add_sched_data(task, "wakeup_preempt_rt curr",
              RT_SCHED_CLASS);
992     }
993     return 0;
994 }
995
996 static int __kprobes kp_enqueue_task_fair_pre(struct kprobe *p,
       struct pt_regs *regs)
```

```
997 {
998     struct task_struct *task = (struct task_struct *)
        p_regs_get_second_arg(regs);
999     if (task && task->pid == target_pid)
1000    {
1001        printk(KERN_INFO "enqueue_task_fair() %d", task->pid);
1002        add_sched_data(task, "enqueue_task_fair",
            CFS_SCHED_CLASS);
1003    }
1004    return 0;
1005 }
1006
1007 static int __kprobes kp_enqueue_task_rt_pre(struct kprobe *p,
    struct pt_regs *regs)
1008 {
1009    struct task_struct *task = (struct task_struct *)
       p_regs_get_second_arg(regs);
1010    if (task)
1011    printk(KERN_INFO "enqueue_task_rt() %d, %s", task->pid,
       task->comm);
1012    if (task && task->pid == target_pid)
1013    {
1014        printk(KERN_INFO "enqueue_task_rt() %d, %s", task->pid,
           task->comm);
1015        add_sched_data(task, "enqueue_task_rt", RT_SCHED_CLASS);
1016    }
1017    return 0;
1018 }
1019
1020 static int __kprobes kp_enqueue_task_dl_pre(struct kprobe *p,
    struct pt_regs *regs)
1021 {
1022    struct task_struct *task = (struct task_struct *)
       p_regs_get_second_arg(regs);
1023    if (task)
```

```
1024     printk(KERN_INFO "enqueue_task_dl() %d, %s", task->pid,
            task->comm);
1025     if (task && task->pid == target_pid)
1026     {
1027         printk(KERN_INFO "enqueue_task_dl() %d", task->pid);
1028         add_sched_data(task, "enqueue_task_dl", DL_SCHED_CLASS);
1029     }
1030     return 0;
1031 }
1032
1033 static int __kprobes kp_dequeue_task_fair_pre(struct kprobe *p,
       struct pt_regs *regs)
1034 {
1035     struct task_struct *task = (struct task_struct *)
            p_regs_get_second_arg(regs);
1036     if (task && task->pid == target_pid)
1037     {
1038         printk(KERN_INFO "dequeue_task_fair() %d", task->pid);
1039         add_sched_data(task, "dequeue_task_fair",
               CFS_SCHED_CLASS);
1040     }
1041     return 0;
1042 }
1043
1044 static int __kprobes kp_dequeue_task_rt_pre(struct kprobe *p,
       struct pt_regs *regs)
1045 {
1046     struct task_struct *task = (struct task_struct *)
            p_regs_get_second_arg(regs);
1047     if (task && task->pid == target_pid)
1048     {
1049         printk(KERN_INFO "dequeue_task_rt() %d", task->pid);
1050         add_sched_data(task, "dequeue_task_rt", RT_SCHED_CLASS);
1051     }
1052     return 0;
```

```
1053 }
1054
1055 static int __kprobes kp_dequeue_task_dl_pre(struct kprobe *p,
        struct pt_regs *regs)
1056 {
1057     struct task_struct *task = (struct task_struct *)
            p_regs_get_second_arg(regs);
1058     if (task && task->pid == target_pid)
1059     {
1060         printk(KERN_INFO "dequeue_task_dl() %d", task->pid);
1061         add_sched_data(task, "dequeue_task_dl", DL_SCHED_CLASS);
1062     }
1063     return 0;
1064 }
1065
1066 static int __kprobes kp_put_prev_task_fair_pre(struct kprobe *p,
        struct pt_regs *regs)
1067 {
1068     struct task_struct *task = (struct task_struct *)
            p_regs_get_second_arg(regs);
1069     if (task && task->pid == target_pid)
1070     {
1071         printk(KERN_INFO "put_prev_task_fair() %d", task->pid);
1072         add_sched_data(task, "put_prev_task_fair",
                CFS_SCHED_CLASS);
1073     }
1074     return 0;
1075 }
1076
1077 static int __kprobes kp_put_prev_task_rt_pre(struct kprobe *p,
        struct pt_regs *regs)
1078 {
1079     struct task_struct *task = (struct task_struct *)
            p_regs_get_second_arg(regs);
1080     if (task && task->pid == target_pid)
```

```
1081     {
1082         printk(KERN_INFO "put_prev_task_rt() %d", task->pid);
1083         add_sched_data(task, "put_prev_task_rt", RT_SCHED_CLASS);
1084     }
1085     return 0;
1086 }
1087
1088 static int __kprobes kp_put_prev_task_dl_pre(struct kprobe *p,
       struct pt_regs *regs)
1089 {
1090     struct task_struct *task = (struct task_struct *)
          p_regs_get_second_arg(regs);
1091     if (task && task->pid == target_pid)
1092     {
1093         printk(KERN_INFO "put_prev_task_dl() %d", task->pid);
1094         add_sched_data(task, "put_prev_task_dl", DL_SCHED_CLASS);
1095     }
1096     return 0;
1097 }
1098
1099 static int __kprobes kp_set_next_task_fair_pre(struct kprobe *p,
       struct pt_regs *regs)
1100 {
1101     struct task_struct *task = (struct task_struct *)
          p_regs_get_second_arg(regs);
1102     if (task && task->pid == target_pid)
1103     {
1104         printk(KERN_INFO "set_next_task_fair() %d", task->pid);
1105         add_sched_data(task, "set_next_task_fair",
              CFS_SCHED_CLASS);
1106     }
1107     return 0;
1108 }
1109
```

```
1110 static int __kprobes kp_set_next_task_rt_pre(struct kprobe *p,
     struct pt_regs *regs)
1111 {
1112     struct task_struct *task = (struct task_struct *)
          p_regs_get_second_arg(regs);
1113     if (task && task->pid == target_pid)
1114     {
1115         printk(KERN_INFO "set_next_task_rt() %d", task->pid);
1116         add_sched_data(task, "set_next_task_rt", RT_SCHED_CLASS);
1117     }
1118     return 0;
1119 }
1120
1121 static int __kprobes kp_set_next_task_dl_pre(struct kprobe *p,
     struct pt_regs *regs)
1122 {
1123     struct task_struct *task = (struct task_struct *)
          p_regs_get_second_arg(regs);
1124     if (task && task->pid == target_pid)
1125     {
1126         printk(KERN_INFO "set_next_task_dl() %d", task->pid);
1127         add_sched_data(task, "set_next_task_dl", DL_SCHED_CLASS);
1128     }
1129     return 0;
1130 }
1131
1132 static int __kprobes kp_set_next_task_idle_pre(struct kprobe *p,
     struct pt_regs *regs)
1133 {
1134     struct task_struct *task = (struct task_struct *)
          p_regs_get_second_arg(regs);
1135     if (task && task->pid == target_pid)
1136     {
1137         printk(KERN_INFO "set_next_task_idle() %d", task->pid);
```

```
1138            add_sched_data(task, "set_next_task_idle",
                    IDLE_SCHED_CLASS);
1139        }
1140        return 0;
1141    }
1142
1143    static int __kprobes kp_yield_task_fair_pre(struct kprobe *p,
            struct pt_regs *regs)
1144    {
1145        struct my_rq *rq = NULL;
1146        struct task_struct *task = NULL;
1147        rq = (struct my_rq *) p_regs_get_first_arg(regs);
1148        if (rq)
1149        {
1150            task = (struct task_struct *) rq->curr;
1151            if (task && task->pid == target_pid)
1152            {
1153                printk(KERN_INFO "yield_task_fair() %d", task->pid);
1154                add_sched_data(task, "yield_task_fair",
                        CFS_SCHED_CLASS);
1155            }
1156        }
1157        return 0;
1158    }
1159
1160    static int __kprobes kp_yield_task_rt_pre(struct kprobe *p,
            struct pt_regs *regs)
1161    {
1162        struct my_rq *rq = NULL;
1163        struct task_struct *task = NULL;
1164        rq = (struct my_rq *) p_regs_get_first_arg(regs);
1165        if (rq)
1166        {
1167            task = (struct task_struct *) rq->curr;
1168            if (task && task->pid == target_pid)
```

```
1169         {
1170             printk(KERN_INFO "yield_task_rt() %d", task->pid);
1171             add_sched_data(task, "yield_task_rt", RT_SCHED_CLASS);
1172         }
1173     }
1174     return 0;
1175 }
1176
1177 static int __kprobes kp_yield_task_dl_pre(struct kprobe *p,
        struct pt_regs *regs)
1178 {
1179     struct my_rq *rq = NULL;
1180     struct task_struct *task = NULL;
1181     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1182     if (rq)
1183     {
1184         task = (struct task_struct *) rq->curr;
1185         if (task && task->pid == target_pid)
1186         {
1187             printk(KERN_INFO "yield_task_dl() %d", task->pid);
1188             add_sched_data(task, "yield_task_dl", DL_SCHED_CLASS);
1189         }
1190     }
1191     return 0;
1192 }
1193
1194 static int __kprobes kp_task_tick_fair_pre(struct kprobe *p,
        struct pt_regs *regs)
1195 {
1196     struct task_struct *task = NULL;
1197     task = (struct task_struct *) p_regs_get_second_arg(regs);
1198     if (task && task->pid == target_pid)
1199     {
1200         printk(KERN_INFO "before task_tick_fair() %d", task->pid);
```

```
1201         add_sched_data(task, "task_tick_fair before",
                CFS_SCHED_CLASS);
1202     }
1203     return 0;
1204 }
1205
1206 static int __kprobes kp_task_tick_rt_pre(struct kprobe *p, struct
        pt_regs *regs)
1207 {
1208     struct task_struct *task = NULL;
1209     task = (struct task_struct *) p_regs_get_second_arg(regs);
1210     if (task && task->pid == target_pid)
1211     {
1212         printk(KERN_INFO "before task_tick_rt() %d", task->pid);
1213         add_sched_data(task, "task_tick_rt before",
                RT_SCHED_CLASS);
1214     }
1215     return 0;
1216 }
1217
1218 static int __kprobes kp_task_tick_idle_pre(struct kprobe *p,
        struct pt_regs *regs)
1219 {
1220     struct task_struct *task = NULL;
1221     task = (struct task_struct *) p_regs_get_second_arg(regs);
1222     if (task && task->pid == target_pid)
1223     {
1224         printk(KERN_INFO "task_tick_idle() %d", task->pid);
1225         add_sched_data(task, "task_tick_idle", IDLE_SCHED_CLASS);
1226     }
1227     return 0;
1228 }
1229
1230 static int __kprobes kp_task_tick_dl_pre(struct kprobe *p, struct
        pt_regs *regs)
```

```
1231 {
1232     struct task_struct *task = NULL;
1233     task = (struct task_struct *) p_regs_get_second_arg(regs);
1234     if (task && task->pid == target_pid)
1235     {
1236         printk(KERN_INFO "before task_tick_dl() %d", task->pid);
1237         add_sched_data(task, "task_tick_dl before",
                DL_SCHED_CLASS);
1238     }
1239     return 0;
1240 }
1241
1242 static void __kprobes kp_task_tick_fair_post(struct kprobe *p,
       struct pt_regs *regs, unsigned long flags)
1243 {
1244     struct task_struct *task = NULL;
1245     task = (struct task_struct *) p_regs_get_second_arg(regs);
1246     if (task && task->pid == target_pid)
1247     {
1248         printk(KERN_INFO "after task_tick_fair() %d", task->pid);
1249         add_sched_data(task, "task_tick_fair after",
                CFS_SCHED_CLASS);
1250     }
1251 }
1252
1253 static void __kprobes kp_task_tick_rt_post(struct kprobe *p,
       struct pt_regs *regs, unsigned long flags)
1254 {
1255     struct task_struct *task = NULL;
1256     task = (struct task_struct *) p_regs_get_second_arg(regs);
1257     if (task)
1258     printk(KERN_INFO "task_tick_rt() %d", task->pid);
1259     if (task && task->pid == target_pid)
1260     {
1261         printk(KERN_INFO "after task_tick_rt() %d", task->pid);
```

```
1262            add_sched_data(task, "task_tick_rt after",
                    RT_SCHED_CLASS);
1263        }
1264    }
1265
1266    static void __kprobes kp_task_tick_dl_post(struct kprobe *p,
            struct pt_regs *regs, unsigned long flags)
1267    {
1268        struct task_struct *task = NULL;
1269        task = (struct task_struct *) p_regs_get_second_arg(regs);
1270        if (task)
1271        printk(KERN_INFO "task_tick_dl() %d", task->pid);
1272        if (task && task->pid == target_pid)
1273        {
1274            printk(KERN_INFO "after task_tick_dl() %d", task->pid);
1275            add_sched_data(task, "task_tick_dl after",
                    DL_SCHED_CLASS);
1276        }
1277    }
1278
1279    #define TASK_ON_RQ_QUEUED   1
1280
1281    static int __kprobes kp_prio_changed_fair_pre(struct kprobe *p,
            struct pt_regs *regs)
1282    {
1283        struct my_rq *rq = NULL;
1284        struct task_struct *task = NULL;
1285        rq = (struct my_rq *) p_regs_get_first_arg(regs);
1286        if (!rq)
1287        return 0;
1288        task = (struct task_struct *) p_regs_get_second_arg(regs);
1289        if (!task)
1290        return 0;
1291        if (task->pid == target_pid && task->on_rq ==
                TASK_ON_RQ_QUEUED && rq->cfs.nr_running != 1)
```

```
1292     {
1293         printk(KERN_INFO "prio_changed_fair() %d", task->pid);
1294         add_sched_data(task, "prio_changed_fair",
                CFS_SCHED_CLASS);
1295     }
1296     return 0;
1297 }
1298
1299 static int __kprobes kp_prio_changed_rt_pre(struct kprobe *p,
        struct pt_regs *regs)
1300 {
1301     struct my_rq *rq = NULL;
1302     struct task_struct *task = NULL;
1303     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1304     if (!rq)
1305     return 0;
1306     task = (struct task_struct *) p_regs_get_second_arg(regs);
1307     if (!task)
1308     return 0;
1309     if (task->pid == target_pid && task->on_rq ==
            TASK_ON_RQ_QUEUED)
1310     {
1311         printk(KERN_INFO "prio_changed_rt() %d", task->pid);
1312         add_sched_data(task, "prio_changed_rt", RT_SCHED_CLASS);
1313     }
1314     return 0;
1315 }
1316
1317 static int __kprobes kp_prio_changed_dl_pre(struct kprobe *p,
        struct pt_regs *regs)
1318 {
1319     struct my_rq *rq = NULL;
1320     struct task_struct *task = NULL;
1321     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1322     if (!rq)
```

```
1323     return 0;
1324     task = (struct task_struct *) p_regs_get_second_arg(regs);
1325     if (!task)
1326     return 0;
1327     if (task->pid == target_pid && task->on_rq ==
           TASK_ON_RQ_QUEUED)
1328     {
1329         printk(KERN_INFO "prio_changed_dl() %d", task->pid);
1330         add_sched_data(task, "prio_changed_dl", DL_SCHED_CLASS);
1331     }
1332     return 0;
1333 }
1334
1335 static int __kprobes kp_update_curr_fair_pre(struct kprobe *p,
       struct pt_regs *regs)
1336 {
1337     struct my_rq *rq = NULL;
1338     struct task_struct *task = NULL;
1339     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1340     if (!rq)
1341     return 0;
1342     task = rq->curr;
1343     if (task && task->pid == target_pid)
1344     {
1345         printk(KERN_INFO "update_curr_fair() %d", task->pid);
1346         add_sched_data(task, "update_curr_fair", CFS_SCHED_CLASS);
1347     }
1348     return 0;
1349 }
1350
1351 static int __kprobes kp_update_curr_rt_pre(struct kprobe *p,
       struct pt_regs *regs)
1352 {
1353     struct my_rq *rq = NULL;
1354     struct task_struct *task = NULL;
```

```
1355     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1356     if (!rq)
1357     return 0;
1358     task = rq->curr;
1359     if (task && task->pid == target_pid)
1360     {
1361         printk(KERN_INFO "update_curr_rt() %d", task->pid);
1362         add_sched_data(task, "update_curr_rt", RT_SCHED_CLASS);
1363     }
1364     return 0;
1365 }
1366
1367 static int __kprobes kp_update_curr_idle_pre(struct kprobe *p,
        struct pt_regs *regs)
1368 {
1369     struct my_rq *rq = NULL;
1370     struct task_struct *task = NULL;
1371     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1372     if (!rq)
1373     return 0;
1374     task = rq->curr;
1375     if (task)
1376     printk(KERN_INFO "update_curr_idle() %d, %s", task->pid,
            task->comm);
1377     if (task && task->pid == target_pid)
1378     {
1379         printk(KERN_INFO "update_curr_idle() %d", task->pid);
1380         add_sched_data(task, "update_curr_idle",
                IDLE_SCHED_CLASS);
1381     }
1382     return 0;
1383 }
1384
1385 static int __kprobes kp_update_curr_dl_pre(struct kprobe *p,
        struct pt_regs *regs)
```

```
1386 {
1387     struct my_rq *rq = NULL;
1388     struct task_struct *task = NULL;
1389     rq = (struct my_rq *) p_regs_get_first_arg(regs);
1390     if (!rq)
1391     return 0;
1392     task = rq->curr;
1393     if (task && task->pid == target_pid)
1394     {
1395         printk(KERN_INFO "update_curr_dl() %d", task->pid);
1396         add_sched_data(task, "update_curr_dl", DL_SCHED_CLASS);
1397     }
1398     return 0;
1399 }
1400
1401 static struct kprobe kp_hooks[] = {
1402     {
1403         .symbol_name = "check_preempt_wakeup_fair",
1404         .post_handler = kp_check_preempt_fair_post,
1405     },
1406     {
1407         .symbol_name = "wakeup_preempt_dl",
1408         .pre_handler = kp_wakeup_preempt_dl_pre,
1409     },
1410     {
1411         .symbol_name = "wakeup_preempt_rt",
1412         .pre_handler = kp_wakeup_preempt_rt_pre,
1413     },
1414     {
1415         .symbol_name = "wakeup_preempt_idle",
1416         .pre_handler = kp_wakeup_preempt_idle_pre,
1417     },
1418     {
1419         .symbol_name = "enqueue_task_fair",
1420         .pre_handler = kp_enqueue_task_fair_pre,
```

```
1421        },
1422        {
1423            .symbol_name = "enqueue_task_rt",
1424            .pre_handler = kp_enqueue_task_rt_pre,
1425        },
1426        {
1427            .symbol_name = "enqueue_task_dl",
1428            .pre_handler = kp_enqueue_task_dl_pre,
1429        },
1430        {
1431            .symbol_name = "dequeue_task_fair",
1432            .pre_handler = kp_dequeue_task_fair_pre,
1433        },
1434        {
1435            .symbol_name = "dequeue_task_rt",
1436            .pre_handler = kp_dequeue_task_rt_pre,
1437        },
1438        {
1439            .symbol_name = "dequeue_task_dl",
1440            .pre_handler = kp_dequeue_task_dl_pre,
1441        },
1442        {
1443            .symbol_name = "put_prev_task_fair",
1444            .pre_handler = kp_put_prev_task_fair_pre,
1445        },
1446        {
1447            .symbol_name = "put_prev_task_rt",
1448            .pre_handler = kp_put_prev_task_rt_pre,
1449        },
1450        {
1451            .symbol_name = "put_prev_task_dl",
1452            .pre_handler = kp_put_prev_task_dl_pre,
1453        },
1454        {
1455            .symbol_name = "set_next_task_fair",
```

```
1456            .pre_handler = kp_set_next_task_fair_pre,
1457        },
1458        {
1459            .symbol_name = "set_next_task_rt",
1460            .pre_handler = kp_set_next_task_rt_pre,
1461        },
1462        {
1463            .symbol_name = "set_next_task_dl",
1464            .pre_handler = kp_set_next_task_dl_pre,
1465        },
1466        {
1467            .symbol_name = "set_next_task_idle",
1468            .pre_handler = kp_set_next_task_idle_pre,
1469        },
1470        {
1471            .symbol_name = "yield_task_fair",
1472            .pre_handler = kp_yield_task_fair_pre,
1473        },
1474        {
1475            .symbol_name = "yield_task_rt",
1476            .pre_handler = kp_yield_task_rt_pre,
1477        },
1478        {
1479            .symbol_name = "yield_task_dl",
1480            .pre_handler = kp_yield_task_dl_pre,
1481        },
1482        {
1483            .symbol_name = "task_tick_fair",
1484            .pre_handler = kp_task_tick_fair_pre,
1485            .post_handler = kp_task_tick_fair_post,
1486        },
1487        {
1488            .symbol_name = "task_tick_rt",
1489            .pre_handler = kp_task_tick_rt_pre,
1490            .post_handler = kp_task_tick_rt_post,
```

```
1491        },
1492        {
1493            .symbol_name = "task_tick_dl",
1494            .pre_handler = kp_task_tick_dl_pre,
1495            .post_handler = kp_task_tick_dl_post,
1496        },
1497        {
1498            .symbol_name = "task_tick_idle",
1499            .pre_handler = kp_task_tick_idle_pre,
1500        },
1501        {
1502            .symbol_name = "prio_changed_fair",
1503            .pre_handler = kp_prio_changed_fair_pre,
1504        },
1505        {
1506            .symbol_name = "prio_changed_rt",
1507            .pre_handler = kp_prio_changed_rt_pre,
1508        },
1509        {
1510            .symbol_name = "prio_changed_dl",
1511            .pre_handler = kp_prio_changed_dl_pre,
1512        },
1513        {
1514            .symbol_name = "update_curr_fair",
1515            .pre_handler = kp_update_curr_fair_pre,
1516        },
1517        {
1518            .symbol_name = "update_curr_rt",
1519            .pre_handler = kp_update_curr_rt_pre,
1520        },
1521        {
1522            .symbol_name = "update_curr_dl",
1523            .pre_handler = kp_update_curr_dl_pre,
1524        },
1525        {
```

```
1526            .symbol_name = "update_curr_idle",
1527            .pre_handler = kp_update_curr_idle_pre,
1528        },
1529 };
1530
1531 #define KHOOK(_name, _function, _original)   \
1532 {                       \
1533     .name = (_name),      \
1534     .function = (_function),      \
1535     .original = (_original),      \
1536 }
1537
1538 static struct ftrace_hook hooked_functions[] = {
1539     KHOOK("yield_to_task_fair", fh_yield_to_task_fair,
          &real_yield_to_task_fair),
1540     KHOOK("__pick_next_task_fair", fh_pick_next_task_fair,
          &real_pick_next_task_fair),
1541     KHOOK("pick_next_task_rt", fh_pick_next_task_rt,
          &real_pick_next_task_rt),
1542     KHOOK("pick_next_task_dl", fh_pick_next_task_dl,
          &real_pick_next_task_dl),
1543     KHOOK("pick_next_task_idle", fh_pick_next_task_idle,
          &real_pick_next_task_idle),
1544 };
1545
1546 int install_hooks()
1547 {
1548     int i = 0;
1549     int ret = fh_install_hooks(hooked_functions,
          ARRAY_SIZE(hooked_functions));
1550     if (ret < 0)
1551     return ret;
1552     for (;ret == 0 && i < ARRAY_SIZE(kp_hooks); i++)
1553     {
1554         ret = register_kprobe(&kp_hooks[i]);
```

```
1555     }
1556
1557     if (ret < 0)
1558     {
1559         fh_remove_hooks(hooked_functions,
             ARRAY_SIZE(hooked_functions));
1560         i--;
1561         for (;i >= 0; i--)
1562         unregister_kprobe(&kp_hooks[i]);
1563     }
1564     return ret;
1565 }
1566
1567 void remove_hooks()
1568 {
1569     fh_remove_hooks(hooked_functions,
        ARRAY_SIZE(hooked_functions));
1570     for (int i = 0; i < ARRAY_SIZE(kp_hooks); i++)
1571     {
1572         unregister_kprobe(&kp_hooks[i]);
1573         printk(KERN_INFO "unregister kprobe %d", i);
1574     }
1575     printk(KERN_INFO "unregister kprobes");
1576 }
1577
1578 static int __init md_init(void)
1579 {
1580     printk(KERN_INFO "Initializing module\n");
1581     if (target_pid == 0) {
1582         printk(KERN_INFO "Error: Target PID is not specified\n");
1583         return -EINVAL;
1584     }
1585     printk(KERN_INFO "module: target pid is %d\n", target_pid);
1586     int ret;
1587     init_process_sched_data_list();
```

```
1588     if ((seq_file = proc_create(FILENAME, 0666, NULL, &fops)) ==
             NULL)
1589     {
1590         printk(KERN_ERR "+seq create file error\n");
1591         free_process_sched_data_list();
1592         return -ENOMEM;
1593     }
1594     int err = install_hooks();
1595     if (err) {
1596         remove_proc_entry(FILENAME, NULL);
1597         free_process_sched_data_list();
1598         printk(KERN_INFO "module error\n");
1599         return err;
1600     }
1601     printk(KERN_INFO "Module: loaded\n");
1602     return 0;
1603 }
1604
1605 static void __exit md_exit(void)
1606 {
1607     printk(KERN_INFO "Module: Unloading module\n");
1608     remove_hooks();
1609     remove_proc_entry(FILENAME, NULL);
1610     free_process_sched_data_list();
1611     printk(KERN_INFO "Module: removed\n");
1612 }
1613
1614 module_init(md_init);
1615 module_exit(md_exit);
```