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

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

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

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Мониторинг сетевой подсистемы Linux»

Студент **ИУ7-71Б** _____ **Волков Г. В.**

Руководитель КР _____ **Рязанова Н. Ю.**

Рекомендуемая оценка _____

2023 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 ИУ-7
_____ И. В. Рудаков
«25» декабпя 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы

по теме

«Мониторинг сетевой подсистемы Linux»

Студент группы **ИУ7-71Б**

Волков Георгий Валерьевич

Направленность КР

учебная

Источник тематики

НИР кафедры

График выполнения НИР: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание

Провести анализ сетевой подсистемы Linux. Разработать загружаемый модуль ядра, предоставляющий пользователю возможность получения информации о сетевой подсистеме.

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на **12-20** листах формата А4.

Дата выдачи задания «25» декабря 2023 г.

Руководитель КР	_____	**Рязанова Н. Ю.**
	(Подпись, дата)	(Фамилия И. О.)
Студент	_____	**Волков Г. В.**
	(Подпись, дата)	(Фамилия И. О.)

РЕФЕРАТ

Расчётно–пояснительная записка 39 с., 7 рис., 12 источн.

ОПЕРАЦИОННЫЕ СИСТЕМЫ, ЗАГРУЖАЕМЫЙ МОДУЛЬ ЯДРА, СЕТЕВАЯ ПОДСИСТЕМА LINUX

Цель работы — разработать загружаемый модуль ядра, предоставляющий информацию о работе сетевой подсистемы Linux.

В процессе работы были проанализированы приём и отправка сетевого кадра и реализован модуль выводящий некоторую информацию о работе системы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В 2023 году операционными системами на основе Linux пользуются 47% разработчиков, 40% веб–сайтов, 85% смартфонов и 96% серверов [1]. Эти люди и устройства постоянно генерируют большое количество сетевого трафика, который нужно успевать обрабатывать.

Получением, отправкой и пересылкой трафика занимается сетевая подсистема Linux. Её мониторинг позволит выявить узкие места и правильно настроить её компоненты.

Целью данной курсовой работы — разработка загружаемого модуля ядра, предоставляющего пользователю информацию о работе сетевой подсистемы Linux.

Для достижения поставленной в работе цели предстоит решить следующие задачи:

— произвести анализ функций и структур, используемых для обработки сетевых кадров;

— разработать загружаемый модуль ядра, предоставляющий информацию о работе сетевой подсистемы;

— реализовать загружаемый модуль ядра.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу необходимо разработать загружаемый модуль, предоставляющий пользователю информацию о работе сетевой подсистемы Linux. Для решения поставленной задачи необходимо:

— произвести анализ функций и структур, используемых для обработки сетевых кадров;

— разработать загружаемый модуль ядра, предоставляющий информацию о работе сетевой подсистемы;

— реализовать модуль ядра;

— протестировать работу реализованного загружаемого модуля.

1.2 Взаимодействие сетевой карты и сетевой подсистемы

Почти все устройства взаимодействуют с ядром одним из двух способов: опрос и прерывания. Также на практике может применяться комбинация этих методов [2].

При **опросе** ядро постоянно проверяет, есть ли у устройства какие–то данные для передачи. Оно может делать это, например, постоянно считывая регистр памяти на устройстве или переодически, по истечению таймера, проводить проверку. Такой подход требует большого количества системных ресурсов и редко применяется на практике.

При использовании **прерываний** устройство генерирует аппаратный сигнал при возникновении определённых событий. Каждое прерывание запускает функцию, называемую обработчиком прерываний, которая должна

быть совместима с устройством, следовательно, она регистрируется драйвером устройства при его загрузке. Для идентификации обработчика ядру нужны как номер IRQ, так и идентификатор устройства. Это нужно, так как IRQ может совместно использоваться несколькими устройствами при определённых условиях.

При прерывании сетевая карта может сообщить своему драйверу несколько разных вещей. Среди них:

— получение кадра;

— сбой передачи;

— передача DMA успешно завершена — получив кадр для отправки, буфер, в котором он хранится, освобождается драйвером, как только кадр загружается в память сетевой карты для передачи. При синхронных передачах (без DMA) драйвер сразу узнает, когда кадр был загружен на сетевую карту. Но при использовании DMA, который использует асинхронные передачи, драйверу устройства необходимо дождаться явного прерывания от сетевой карты;

— устройство имеет достаточно памяти для обработки новой передачи — драйвер сетевого устройства обычно отключает передачу, останавливая очередь на выход, когда в этой очереди недостаточно свободного места.

Этот метод представляет собой наилучший вариант при низких нагрузках на трафик. Но он плохо работает при высокой нагрузке, потому что обработка прерываний для обслуживания каждого кадра может занять большую часть времени.

Большое количество драйверов обрабатывают сразу несколько кадров при прерывании. Обработчик, зарегистрированный драйвером, загружает кадры и помещает их в очередь ввода ядра. Подобным образом функционирует NAPI.

Прерывания, управляемые таймером это метод, который является усовершенствованием предыдущих. Вместо того, чтобы устройство асинхронно уведомляло драйвер о приёме кадра, прерывания генерируются с определённым интервалом. Затем обработчик проверит, поступили ли какие-либо кадры после предыдущего прерывания, и обработает их все за один раз.

1.3 Обработка прерываний

Всякий раз, когда процессор получает прерывании, он вызывает обработчик, связанный с этим прерыванием. Во время выполнения обработчика, которой выполняется в контексте прерывания, другие прерывания отключаются для этого процессора. Это означает, что если процессор занят обслуживанием одного прерывания, он не может обслуживать другие. Он также не может выполнять какой–либо другой процесс. Такой выбор дизайна помогает снизить вероятность возникновения условий гонки. Однако такие жёсткие ограничения на работу процессора серьёзно влияют на производительность системы. Следовательно, работа, выполняемая обработчиками прерываний, должна быть как можно более быстрой. Объем работы обработчика зависит от типа события, иногда нужно просто сохранить код нажатой клавиши, а в другом случае действия не являются тривиальными, и их выполнение может потребовать много процессорного времени. У драйверов сетевых устройств относительно сложная работа: им нужно выделить буфер (sk_buff), скопировать в него полученные данные, инициализировать несколько параметров в структуре буфера для обработчиков протоколов более высокого уровня и передать дальше по цепочке обслуживания [2].

По этой причине современные обработчики прерываний делятся на верхнюю и нижнюю половины. Верхняя половина состоит из всего, что должно быть выполнено перед освобождением процессора, как правило это загрузки данных, необходимых для дальнейшей обработки. Нижняя половина содержит все остальное, то есть выполняет основную часть работы по обработке прерывания. Нижнюю половину можно определить как асинхронный запрос на выполнение определённой функции. Следующая модель обработки прерываний позволяет ядру отключать прерывания на гораздо меньшее время:

— устройство генерирует сигнал прерывания;

— процессор выполняет верхнюю половину, блокируя прерывания, как правило она делает следующее: сохраняет в оперативной памяти всю информацию, которая позже понадобится нижней половине, планирует на выполнение нижнюю половину и разрешает прерывания;

— позднее выполняется нижняя половина прерывания, содержащая основной объём работы, но уже не в контексте прерывания.

Самым большим улучшением между ядрами 2.2 и 2.4, стало внедрение программных прерываний (softirqs), которые можно рассматривать как многопоточную версию обработчиков нижней половины. Многие softirq могут выполняться конкурентно, но также один и тот же softirq может выполняться конкурентно на разных процессорах. Единственное ограничение на параллелизм заключается в том, что только один экземпляр каждого softirq может выполняться одновременно на процессоре. Есть всего 6 типов softirq:

— HI_SOFTIRQ;

— TIMER_SOFTIRQ;

— NET_TX_SOFTIRQ;

— NET_RX_SOFTIRQ;

— SCSI_SOFTIRQ;

— TASKLET_SOFTIRQ.

Для их выполнения в системе запускаются потоки ksoftirqd, по одному на процессор, которые крутятся в цикле в ожидание поступления работы. При наличии запланированных на выполнение нижних половин прерываний вызывается функция do_softirq, которая и выполняет зарегистрированный обработчик. Сама функция do_softirq, проверив, что сейчас не обрабатываются прерывания, проверяет наличие softirq на выполнение и переходит выполнению обработчика (функция __do_softirq). Проходятся по битовой маске в цикле, определяются softirq требующие выполнения и запускаются зарегистрированные обработчики хранящиеся в массиве softirq_vec [3].

В сетевой подсистеме NET_RX_SOFTIRQ используется для обработки входящего трафика (net_rx_action), а NET_TX_SOFTIRQ исходящего (net_tx_action).

Каждый CPU имеет свою собственную структуру данных для управления входящим и исходящим трафиком. Это структура softnet_data, которая представлена в листинге 1.1 [4].

Листинг 1.1 – Структуры softnet_data

```
1  struct softnet_data {
2      struct list_head    poll_list;
3      struct sk_buff_head process_queue;
4      unsigned int        processed;
5      unsigned int        time_squeeze;
6      #ifdef CONFIG_RPS
7      struct softnet_data *rps_ipi_list;
8      #endif
9      bool            in_net_rx_action;
10     bool            in_napi_threaded_poll;
11     #ifdef CONFIG_NET_FLOW_LIMIT
12     struct sd_flow_limit __rcu *flow_limit;
13     #endif
14     struct Qdisc        *output_queue;
15     struct Qdisc        **output_queue_tailp;
16     struct sk_buff      *completion_queue;
17     #ifdef CONFIG_XFRM_OFFLOAD
18     struct sk_buff_head xfrm_backlog;
19     #endif
20     struct {
21         u16 recursion;
22         u8  more;
23         #ifdef CONFIG_NET_EGRESS
24         u8  skip_txqueue;
25         #endif
26     } xmit;
27     #ifdef CONFIG_RPS
28     unsigned int        input_queue_head
           ____cacheline_aligned_in_smp;
29     call_single_data_t  csd ____cacheline_aligned_in_smp;
30     struct softnet_data *rps_ipi_next;
31     unsigned int        cpu;
32     unsigned int        input_queue_tail;
33     #endif
34     unsigned int        received_rps;
35     unsigned int        dropped;
36     struct sk_buff_head input_pkt_queue;
37     struct napi_struct  backlog;
38     spinlock_t      defer_lock ____cacheline_aligned_in_smp;
39     int         defer_count;
```

```
40      int             defer_ipi_scheduled;
41      struct sk_buff      *defer_list;
42      call_single_data_t  defer_csd;
43  };
```

Структура включает в себя как поля, используемые для приёма, так и поля, используемые для передачи. Не все драйвера используют NAPI, но всем они используют эту структуру. Рассмотрим некоторые поля подробнее:

— poll_list — двунаправленный список NAPI–структур;

— process_queue — очередь кадров обрабатываемая в process_backlog;

— processed — количество обработанных кадров;

— time_squeeze — оличество раз, когда у net_rx_action была работа, но бюджета не хватало либо было достигнуто ограничение по времени, прежде чем работа была завершена;

— in_net_rx_action — флаг о том, что данный экземпляр структуры в текущей момент обрабатывается функцией net_rx_action;

— flow_limit — поле, хранящее данные о ограничении потоков RPS;

— output_queue — очередь, хранящая кадры для отправки;

— completion_queue — список буферов данных, которые были успешно переданы и, следовательно, могут быть освобождены;

— received_rps — количество раз, когда посредством межпроцессорного прерывания будили CPU для обработки пакетов;

— dropped — количество отброшенных кадров;

— input_pkt_queue — очередь, где сохраняются входящие кадры перед обработкой драйвером. Она используется драйверами, не использующими NAPI, или как backlog–очередь. Драйвера с NAPI используют свои собственные частные очереди;

— backlog — NAPI–структура для обработки backlog–очереди.

1.4 Механизм NAPI

New Api (NAPI) был создан в качестве механизма снижения количества прерываний, генерируемых сетевыми устройствами по мере прибытия пакетов. Он позволяет драйверу устройства регистрировать функцию poll, вызываемую подсистемой NAPI для сбора данных [5].

Основная идея реализованная в NAPI заключается в комбинации методов прерывания и опроса. Если новые кадры получены, когда ядро ещё не завершило обработку предыдущих, нет необходимости в генерации новых прерывание, можно просто продолжать обрабатывать все, что находится в очереди ввода устройства (с отключёнными прерываниями для устройства), и повторно включать прерывания, как только очередь опустеет. Таким образом, используются преимущества как прерываний, так и опроса [2]:

— асинхронные события, такие как приём одного или нескольких кадров, обозначаются прерываниями, так что ядру не нужно постоянно проверять, пуста ли очередь входа устройства;

— если в очереди входа устройства что–то осталось, не нужно заново генерировать прерывания и тратить время на их обработку.

Алгоритм использования NAPI драйверами сетевых устройств выглядит так:

— драйвер включает NAPI, изначально тот находится в неактивном состоянии;

— прибывает кадр и сетевая карта напрямую отправляет его в память;

— сетевая карта генерирует IRQ, запуская обработчика прерываний;

— обработчик будит подсистему NAPI с помощью SoftIRQ, которая начинает собирать пакеты вызывая зарегистрированную драйвером функцию poll;

— драйвер отключает последующие генерирования прерываний сетевой картой, чтобы позволить подсистеме NAPI обрабатывать пакеты без помех со стороны устройства;

— когда вся работа выполнена, подсистема NAPI отключается, а генерирование прерываний устройством включается снова.

Этот метод сбора данных позволил уменьшить нагрузку на процессор по сравнению со старым методом, поскольку несколько кадров могут одновременно приниматься без необходимости генерирования IRQ для каждого из них. Драйвер устройства реализует функцию poll и регистрирует её с помощью NAPI.

1.5 Получение данных

Высокоуровневый путь, по которому проходит кадр от прибытия до приёмного буфера сокета выглядит так:

— драйвер загружается и инициализируется;

— кадр прибывает из сети в сетевую карту;

— кадр копируется посредством DMA в кольцевой буфер памяти ядра;

— генерируется аппаратное прерывание, чтобы система узнала о появлении пакета в памяти;

— обработчик вызывает NAPI, чтобы начать цикл опроса (poll loop), если он ещё не начат;

— очищаются те области памяти в кольцевом буфере, в которые были записаны сетевые данные;

— данные передаются для дальнейшей обработки на сетевой уровень в виде sk_buff;

— если включено управление пакетами, или если в сетевой карте есть несколько очередей приёма, то фреймы входящих сетевых данных распределяются по нескольким CPU системы;

— данные передаются дальше по сетевому стеку.

При получении кадра на сетевой карте генерируется прерывание. В самом обработчике вызывается функция napi_schedule, в которую как параметр предаётся napi_struct драйвера. Её код представлен в листинге 1.2 [6].

Листинг 1.2 – Функция _____napi_schedule

```
1  static inline void ____napi_schedule(struct softnet_data *sd,
2  struct napi_struct *napi)
3  {
4      struct task_struct *thread;
5
6      lockdep_assert_irqs_disabled();
7
8      if (test_bit(NAPI_STATE_THREADED, &napi->state)) {
9          thread = READ_ONCE(napi->thread);
10         if (thread) {
11             if (READ_ONCE(thread->__state) != TASK_INTERRUPTIBLE)
12             set_bit(NAPI_STATE_SCHED_THREADED, &napi->state);
13             wake_up_process(thread);
14             return;
15         }
16     }
17
18     list_add_tail(&napi->poll_list, &sd->poll_list);
19     WRITE_ONCE(napi->list_owner, smp_processor_id());
20
21     if (!sd->in_net_rx_action)
22     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
23 }
```

Помимо пробуждения треда обработки NAPI в этой функции в конец очереди poll_list структуры softnet_data добавляется структура napi_struct, код которой представлен в листинге 1.3 [7], драйвера, содержащая информацию, необходимую для обработки пришедших на устройство кадров. Также планируется на выполнение нижняя часть прерывания NET_RX_SOFTIRQ, обработчиком которой является функция net_rx_action. Её код представлен в листинге 1.4 [8].

Листинг 1.3 – Структура napi_struct

```
struct napi_struct {
    struct list_head    poll_list;

    unsigned long       state;
    int         weight;
    int         defer_hard_irqs_count;
    unsigned long       gro_bitmask;
    int         (*poll)(struct napi_struct *, int);
    #ifdef CONFIG_NETPOLL
    int         poll_owner;
    #endif
    int         list_owner;
    struct net_device   *dev;
    struct gro_list     gro_hash[GRO_HASH_BUCKETS];
    struct sk_buff      *skb;
    struct list_head    rx_list; /* Pending GRO_NORMAL skbs */
    int         rx_count; /* length of rx_list */
    unsigned int        napi_id;
    struct hrtimer      timer;
    struct task_struct  *thread;
    struct list_head    dev_list;
    struct hlist_node   napi_hash_node;
};
```

Рассмотрим некоторые поля подробнее:

— poll_list — поддерживает двунаправленный список NAPI–структур с входными кадрами, ожидающими обработки;

— poll — функция опроса, зарегистрированная драйвером;

— weight — максимальное количество кадров, которое может быть обработано за один раз;

— dev — дескриптор сетевого устройства.

Листинг 1.4 – Функция net_rx_action

```
static __latent_entropy void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);
```

```
4       unsigned long time_limit = jiffies +
5       usecs_to_jiffies(READ_ONCE(netdev_budget_usecs));
6       int budget = READ_ONCE(netdev_budget);
7       LIST_HEAD(list);
8       LIST_HEAD(repoll);
9
10      start:
11      sd->in_net_rx_action = true;
12      local_irq_disable();
13      list_splice_init(&sd->poll_list, &list);
14      local_irq_enable();
15
16      for (;;) {
17          struct napi_struct *n;
18
19          skb_defer_free_flush(sd);
20
21          if (list_empty(&list)) {
22              if (list_empty(&repoll)) {
23                  sd->in_net_rx_action = false;
24                  barrier();
25                  if (!list_empty(&sd->poll_list))
26                  goto start;
27                  if (!sd_has_rps_ipi_waiting(sd))
28                  goto end;
29              }
30              break;
31          }
32
33          n = list_first_entry(&list, struct napi_struct, poll_list);
34          budget -= napi_poll(n, &repoll);
35
36          if (unlikely(budget <= 0 ||
37          time_after_eq(jiffies, time_limit))) {
38              sd->time_squeeze++;
39              break;
40          }
41      }
42
43      local_irq_disable();
44
```

```
45      list_splice_tail_init(&sd->poll_list, &list);
46      list_splice_tail(&repoll, &list);
47      list_splice(&list, &sd->poll_list);
48      if (!list_empty(&sd->poll_list))
49      __raise_softirq_irqoff(NET_RX_SOFTIRQ);
50      else
51      sd->in_net_rx_action = false;
52
53      net_rps_action_and_irq_enable(sd);
54      end:;
55  }
```

Функция итерируется по списку структур NAPI, стоящих в очереди текущего CPU. Цикл обработки ограничен объём и временем работы. Таким образом ядро не позволяет обработке пакетов занять все ресурсы процессора. budget — это весь доступный бюджет, который будет разделён на все доступные NAPI–структуры, зарегистрированные на этот CPU. Бюджет является настраиваемой величиной.

Выбрав NAPI–структуру (napi_struct) вызывается функция poll, которая возвращает количество обработанных кадров. Сама функция функция собирает сетевые данные и отправляет их для дальнейшей обработки. Затем это количество вычитается из общего бюджета. Если драйверная функция poll расходует весь свой вес (64), она не должна изменять состояние NAPI и эта структура будет добавлена в конец poll_list.

Выход из цикла net_rx_action будет совершён, если: список poll, зарегистрированный для данного CPU, больше не содержит NAPI–структур, остаток бюджета $<= 0$ или был достигнут временной предел в два jiffies. Если были обработаны не все NAPI–структуры, то тогда заново планируется на выполнение NET_RX_SOFTIRQ. Прежде чем выполнить возврат из net_rx_action вызывается net_rps_action_and_irq_enable. Если включено управление принимаемыми пакетами (RPS) то эта функция пробуждает удалённые CPU, чтобы они начали обрабатывать сетевые данные.

Generic Receive Offloading (GRO) — это программная реализация аппаратной оптимизации, известной как Large Receive Offloading (LRO). Суть обоих механизмов в том, чтобы уменьшить количество пакетов, передаваемых по сетевому стеку, за счёт комбинирования «достаточно похожих» па-

кетов. Это позволяет снизить нагрузку на CPU. Пусть передаётся большой файл, и большинство пакетов содержат чанки данных из этого файла. Вместо отправки по стеку маленьких кадров по одному, входящие кадры можно комбинировать в один большой. А затем уже передавать его по стеку. Таким образом уровни протоколов обрабатывают заголовки одного пакета, при этом передавая данные нескольких маленьких. Но этой оптимизации присуща проблема потери информации. Если какой–то кадр имеет опцию или флаг, то они могут быть потеряны при объединении с другими пакетами.

Функция napi_gro_receive, вызываемая в poll функции драйвера, занимается обработкой сетевых данных для GRO, если включен, и отправкой их дальше по стеку. Большая часть логики находится в функции dev_gro_receive. В самой функции происходи проверка, можно ли объединить пакет с имеющимся потоком. Если пришло время сбросить GRO–пакет, то он передаётся далее по стеку посредством вызова netif_receive_skb. Если пакет не был объединён и в системе меньше MAX_GRO_SKBS (8) GRO-потоков, то в список gro_list NAPI-структуры данного CPU добавляется новая запись. По завершении dev_gro_receive вызывается napi_skb_finish, которая освобождает экземпляры структур, невостребованные по причине слияния пакета, либо для передачи данных по сетевому стеку вызывается netif_receive_skb.

Некоторые сетевые карты на аппаратном уровне поддерживают несколько очередей. Это означает, что входящие пакеты могут напрямую отправляться в разные области памяти, выделенные для каждого очереди. При этом опрос каждой области выполняется с помощью отдельных NAPI-структур. Так что прерывания и пакеты будут обрабатываться несколькими CPU. Этот механизм называется Receive Side Scaling (RSS). Receive Packet Steering (RPS) — это программная реализация RSS. А раз реализовано в коде, то может быть применено для любой сетевой карты, даже если она имеет лишь одну очередь приёма. RPS генерирует для входящих данных хэш, чтобы определить, какой CPU должен их обработать. Затем данные помещаются во входящую очередь (backlog) этого процессора в ожидании последующей обработки. В процессор с backlog передаётся межпроцессорное прерывание (IPI), инициирующее обработку очереди.

netif_receive_skb действует по разному, в зависимости от того, включён ли RPS. Если RPS выключен, то данные просто передаются дальше по

сетевому стеку. Иначе выполняет ряд вычислений чтобы определить, backlog–очередь какого CPU нужно использовать. Для добавления в очередь используется функция enqueue_to_backlog.

Эта функция сначала получает указатель на структуру softnet_data удалённого CPU, содержащую указатель на input_pkt_queue. Если привешен максимальный поток или длинна очереди, то данные отбрасываются. Пусть все проверки пройдены, тогда если очередь пустая: проверяется, запущен ли NAPI на удалённом CPU. Если нет, проверяется, находится ли в очереди на отправку IPI. Если нет, то IPI помещается в очередь, а посредством вызова ____napi_schedule запускается цикл обработки NAPI. Если очередь не пуста, то данные сразу передаются в очередь.

Backlog–очереди каждого CPU используют NAPI так же, как и драйвер устройства. Предоставляется функция poll, используемая для обработки пакетов из контекста SoftIRQ. Как и в случае с драйвером, здесь тоже применяется weight. Эти очереди обслуживаются функцией process_backlog, которая содержит цикл выполняемый до тех пор, пока его вес не будет израсходован или пока не останется больше данных. Данные вынимаются по частям из backlog–очереди и передаются в __netif_receive_skb. Ветвь кода будет такой же, как и в случае с отключённым RPS. Поллер перезапускается посредством вызова ____napi_schedule из enqueue_to_backlog для обработки backlog–очереди.

1.6 Отправка данных

Высокоуровневый путь, по которому проходит пакет при отправке выглядит так:

— данные записываются с помощью системного вызова;

— данные передаются вниз по сетевому стеку, заполняются поля sk_buff;

— выбирается очередь вывода с помощью XPS или хэш–функции;

— данные попадают в соответствующую очередь (qdisc) устройства;

— qdisc либо передаст данные напрямую, если сможет, либо поставит их в очередь для отправки;

— драйвер создаёт необходимые отображения DMA, чтобы устройство могло считывать данные из оперативной памяти;

— драйвер сигнализирует устройству, что данные готовы к передаче;

— устройство извлекает данные из оперативной памяти и передаёт их;

— как только передача завершена, устройство инициирует прерывание, чтобы сигнализировать о завершении передачи.

Linux поддерживает механизм, называемый управлением трафиком. Эта функция позволяет пользователям контролировать передачу пакетов с компьютера. Система управления трафиком содержит несколько различных наборов дисциплин обслуживания (qdisc), которые предоставляют различные функции для управления транспортным потоком [9].

В Linux с каждым интерфейсом связан qdisc по умолчанию. Для сетевого оборудования, поддерживающего только одну очередь передачи, используется pfifo_fast по умолчанию. Сетевое оборудование, поддерживающее несколько очередей передачи, использует mq по умолчанию.

Обработчики протоколов канального уровня, для отправки кадра вызывают функцию dev_queue_xmit. В которой дополнительно обрабатывается sk_buff, чтобы можно было получить доступ к заголовку ethernet и устанавливается приоритет кадра. Далее определяется какую именно очередь передачи использовать, вызовом netdev_pick_tx.

Если имеется более одной очереди, то для определения её номера вызывается ndo_select_queue, реализуема драйвером для более оптимального выбора очереди, или __netdev_pick_tx. В этой функции проверяется была ли очередь закеширована на сокете или нет. Если была, то возвращаем номер этой очереди. Иначе получаем через настройки XPS номер очереди, вызвав get_xps_queue. Если возвращает -1, потому что это ядро не поддерживает XPS, или он не был настроен, или настроенное сопоставление ссылается на недопустимую очередь, вызовется skb_tx_hash, которая вычисляет хеш буфера, который и является номером очереди.

Управление передачей пакетов (XPS) — это функция, которая позволяет пользователю определять, какие процессоры могут обрабатывать операции передачи для каждой очереди, поддерживаемой устройством. Цель этой

функции в основном состоит в том, чтобы избежать блокировки соединений при обработке запросов на передачу.

Получив номер очереди, получаем на неё ссылку и добавляем буфер в очередь вызовом __dev_xmit_skb, если такая операция определена, и переходим к концу функции. Единственными устройствами, которые могут иметь qdisc без очередей, являются устройства обратной связи и туннельные устройства.

В функции __dev_xmit_skb проверятся отключена ли Qdisc, если отключена, то освобождаются данные и возвращается код ошибки. Иначе вызывается qdisc_run для запуска обработки очереди.

В функция sch_direct_xmit если очередь передачи не остановлена, то вызывается dev_hard_start_xmit, которая отвечает за передачу данных с сетевого устройства. Код возврата из этой функции сохраняется и будет проверен далее функцией dev_xmit_complete, чтобы определить, была ли передача успешной. Если данные были отправлены успешно, то возвращается длина очереди. Если был возвращён код NETDEV_TX_BUSY, драйвер сейчас «занят» и не может отправить данные, то вызывается dev_requeue_skb, в которой данные встают в очередь для повторной отправки и планируется сама отправка (__netif_schedule). Иначе дополнительно выводится предупреждение.

Код функции __qdisc_run представлен в листинге 1.5 [10].

Листинг 1.5 – Функция __qdisc_run

```
1  void __qdisc_run(struct Qdisc *q)
2  {
3      int quota = READ_ONCE(dev_tx_weight);
4      int packets;
5
6      while (qdisc_restart(q, &packets)) {
7          quota -= packets;
8          if (quota <= 0) {
9              if (q->flags & TCQ_F_NOLOCK)
10             set_bit(__QDISC_STATE_MISSED, &q->state);
11             else
12             __netif_schedule(q);
13
14             break;
```

```
15          }
16      }
17  }
```

Функция qdisc_restart вызывает dequeue_skb, чтобы получить следующий пакет для передачи. Если очередь пуста, qdisc_restart вернёт значение false, что остановит цикл. Пусть есть данные для передачи, получаются ссылки на блокировку очереди qdisc, связанное с qdisc устройство и очередь передачи и передаются в sch_direct_xmit, чей код возврата и возвращается функцией. То есть в цикле постоянно пытаются передать данные в рамках квоты. Всё что не удалось отправить, планируется для отправки через __netif_schedule. Также dequeue_skb в первую очередь возвращает кадры, которые когда–то пытались отправить, но не получилось, и они вернулись. Например, как при возврате кода NETDEV_TX_BUSY. Всё это ещё выполняется в рамках системного вызова для отправки данных, например, sendmsg.

Код функции __netif_reschedule представлен в листинге 1.6 [11].

Листинг 1.6 – Функция __netif_reschedule

```
1  static void __netif_reschedule(struct Qdisc *q)
2  {
3      struct softnet_data *sd;
4      unsigned long flags;
5
6      local_irq_save(flags);
7      sd = this_cpu_ptr(&softnet_data);
8      q->next_sched = NULL;
9      *sd->output_queue_tailp = q;
10     sd->output_queue_tailp = &q->next_sched;
11     raise_softirq_irqoff(NET_TX_SOFTIRQ);
12     local_irq_restore(flags);
13 }
```

В этой функции выполняются 2 основных действия: добавляется Qdisc в очередь output_queue_tailp на обработку и вызывается NET_TX_SOFTIRQ, обработчиком которой является функция net_tx_action, код которой представлен в листинге 1.7 [12].

Листинг 1.7 – Функция net_tx_action

```
1  static __latent_entropy void net_tx_action(struct softirq_action *h)
2  {
3      struct softnet_data *sd = this_cpu_ptr(&softnet_data);
4
5      if (sd->completion_queue) {
6          struct sk_buff *clist;
7
8          local_irq_disable();
9          clist = sd->completion_queue;
10         sd->completion_queue = NULL;
11         local_irq_enable();
12
13         while (clist) {
14             struct sk_buff *skb = clist;
15
16             clist = clist->next;
17
18             WARN_ON(refcount_read(&skb->users));
19             if (likely(get_kfree_skb_cb(skb)->reason ==
                   SKB_CONSUMED))
20             trace_consume_skb(skb, net_tx_action);
21             else
22             trace_kfree_skb(skb, net_tx_action,
23             get_kfree_skb_cb(skb)->reason);
24
25             if (skb->fclone != SKB_FCLONE_UNAVAILABLE)
26             __kfree_skb(skb);
27             else
28             __napi_kfree_skb(skb,
29             get_kfree_skb_cb(skb)->reason);
30         }
31     }
32
33     if (sd->output_queue) {
34         struct Qdisc *head;
35
36         local_irq_disable();
37         head = sd->output_queue;
38         sd->output_queue = NULL;
39         sd->output_queue_tailp = &sd->output_queue;
```

```
40          local_irq_enable();
41
42          rcu_read_lock();
43
44          while (head) {
45              struct Qdisc *q = head;
46              spinlock_t *root_lock = NULL;
47
48              head = head->next_sched;
49
50              smp_mb__before_atomic();
51
52              if (!(q->flags & TCQ_F_NOLOCK)) {
53                  root_lock = qdisc_lock(q);
54                  spin_lock(root_lock);
55              } else if (unlikely(test_bit(__QDISC_STATE_DEACTIVATED,
56              &q->state))) {
57                  clear_bit(__QDISC_STATE_SCHED, &q->state);
58                  continue;
59              }
60
61              clear_bit(__QDISC_STATE_SCHED, &q->state);
62              qdisc_run(q);
63              if (root_lock)
64              spin_unlock(root_lock);
65          }
66
67          rcu_read_unlock();
68      }
69
70      xfrm_dev_backlog(sd);
71  }
```

Данная функция освобождает completion_queue структуры softnet_data и отправляет данные находящиеся в очередях этой структуры.

completion_queue это просто список sk_buff, ожидающих освобождения. Функция dev_kfree_skb_irq может использоваться для добавления skb в эту очередь. Данные буферы не освобождаются драйвером сразу, так как освобождение памяти может занять время, и есть случаи (например, обработчики hardirq), когда код должен выполняться как можно быстрее и воз-

вращаться. Для освобождения по списку проходят циклом и для каждого элемента вызывается __kfree_skb.

Функция net_tx_action планируется на выполнение в двух случаях: в dev_requeue_skb, когда возникает коллизия или сетевое устройство занято, или в __qdisc_run, когда заканчивается квота. Если что–то есть в output_queue, то начинается обработка цикла. Если очередь требует блокировки то она захватывается, иначе если очередь деактивирован, то снимется бит состояния, запланирован на обработку, и переход к следующему элементу списка. Далее вызывается qdisc_run и снимается блокировка, если она использовалась.

2 Конструкторский раздел

2.1 Последовательность действий ПО

На рисунках 2.1, 2.2 представлены IDEF0–диаграммы описывающие вывод информации о сетевой подсистеме, описывающая работу модуля.

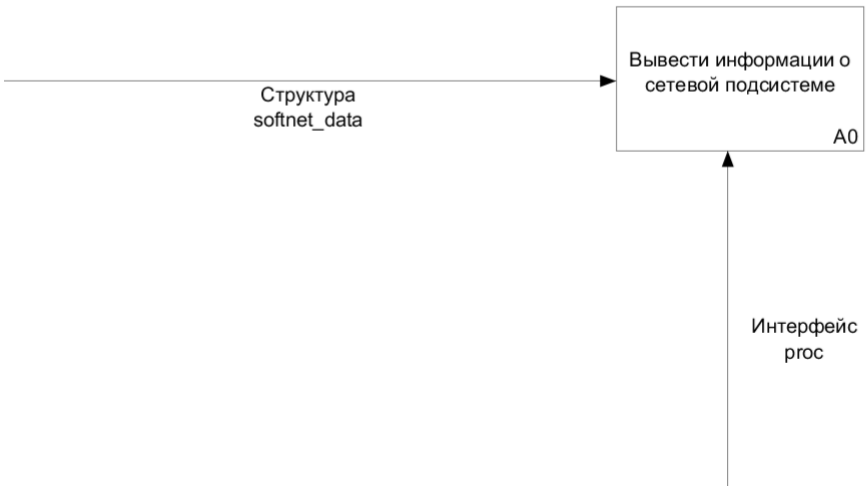

Рисунок 2.1 – Диаграмма описывающая вывод информации о сетевой подсистеме нулевого уровня

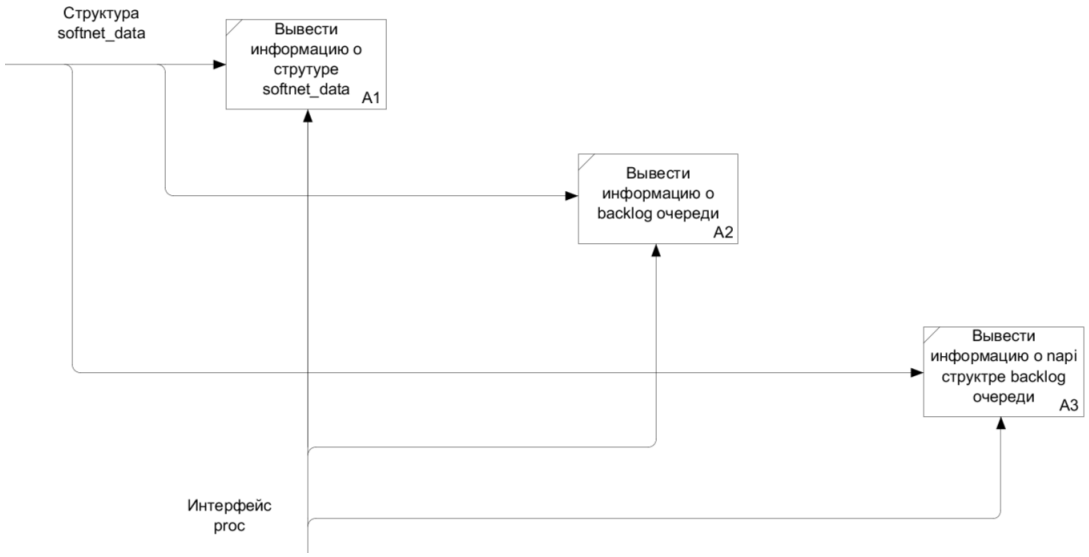

Рисунок 2.2 – Диаграмма описывающая вывод информации о сетевой подсистеме первого уровня

2.2 Алгоритм вывода данных о сетевой подсистеме

На рисунке 2.3 представлена схема алгоритма вывода информации о работе сетевой подсистемы.

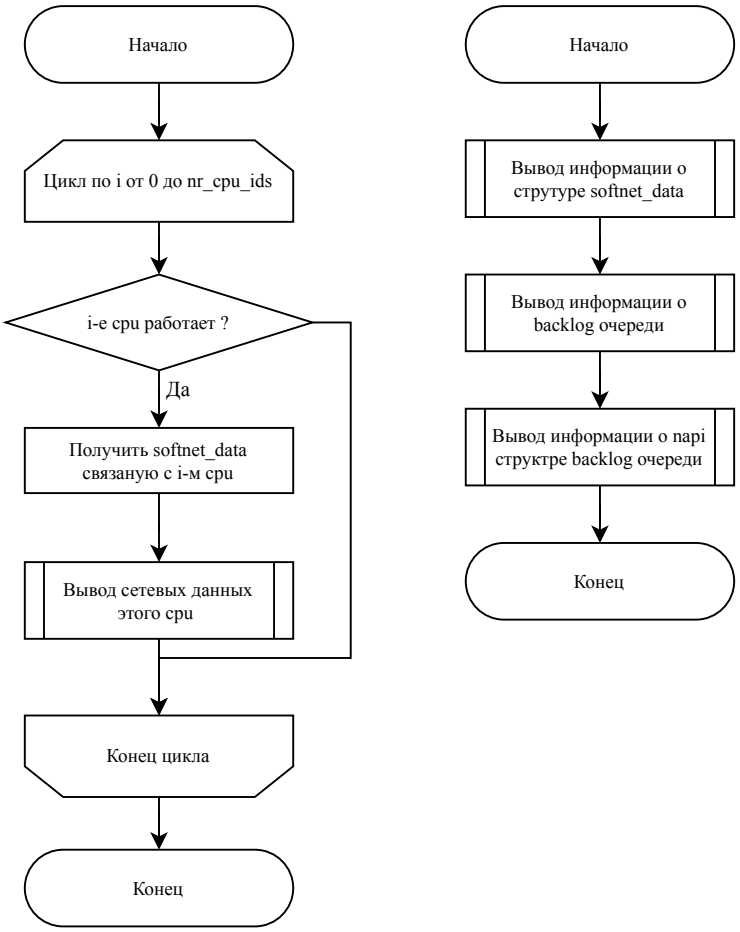

Рисунок 2.3 – Алгоритм вывода информации о работе сетевой подсистемы

На рисунке 2.4 представлена схема алгоритма вывода информации о структуре softnet_data.

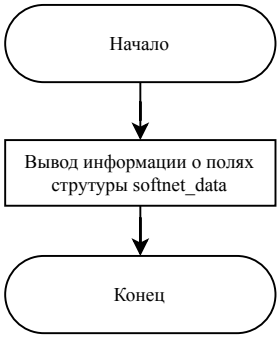

Рисунок 2.4 – Алгоритм вывода информации о структуре softnet_data

На рисунке 2.5 представлена схема алгоритма вывода информации о backlog очереди.

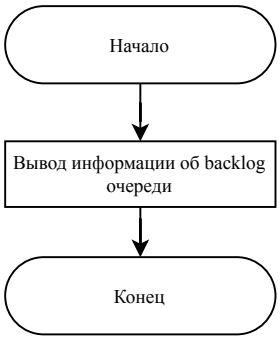

Рисунок 2.5 – Алгоритм вывода информации о backlog очереди

На рисунке 2.6 представлена схема алгоритма вывода информации о структуре napi backlog очереди.

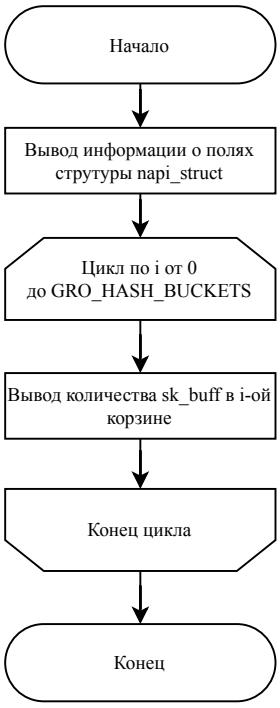

Рисунок 2.6 – Алгоритм вывода информации о структуре napi backlog очереди

3 Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования для реализации поставленной задачи был выбран язык C, так как в нём есть все инструменты для реализации загружаемого модуля ядра и на нём реализовано ядро Linux. Для сборки модуля использовалась утилита make. В качестве среды разработки был выбран Visual Studio Code, так как он является бесплатным и в нём можно настроить пути поиска заголовочных файлов на заголовочные файлы Linux, после чего начинает корректно работа линтер и автодополнение.

3.2 Реализация загружаемого модуля ядра

В листинге 3.1 представлен код функций загрузки и выгрузки модуля. При загрузке создаётся файл в интерфейсе proc и регистрируются функции для работы с ним. Для вывода данных используются sequence файл.

Листинг 3.1 – Функции инциализации и выгрузки модуля

```
1  ssize_t netstat_read(struct file *file, char __user *buf, size_t
       size, loff_t *ppos)
2  {
3      return seq_read(file, buf, size, ppos);
4  }
5
6  int netstat_open(struct inode *inode, struct file *file)
7  {
8      return single_open(file, netstat_show, NULL);
9  }
10
11 int netstat_release(struct inode *inode, struct file *file)
12 {
13     return single_release(inode, file);
14 }
15
16
```

```
17 static struct proc_ops fops = {
18     .proc_read = netstat_read,
19     .proc_open = netstat_open,
20     .proc_release = netstat_release
21 };
22
23 static int __init mod_init(void)
24 {
25     if (!proc_create(FILENAME, 0666, NULL, &fops))
26     {
27         printk(KERN_ERR "%s: file create failed!\n", FILENAME);
28         return -1;
29     }
30
31     printk(KERN_INFO "%s: module loaded\n", FILENAME);
32     return 0;
33 }
34
35 static void __exit mod_exit(void)
36 {
37     remove_proc_entry(FILENAME, NULL);
38     printk(KERN_INFO "%s: module unloaded\n", FILENAME);
39 }
40
41 module_init(mod_init);
42 module_exit(mod_exit);
```

В листинге 3.2 представлен код функций перебора структур softnet_data и вывода информации содержащейся в них.

Листинг 3.2 – Функции перебора и вывода информации о структуре softnet_data

```
1 void print_netdata(struct seq_file *seq, struct softnet_data *sd) {
2     print_softnet_data(seq, sd);
3     print_backlog_data(seq, sd);
4     print_backlog_napi_data(seq, &sd->backlog);
5 }
6
7 int netstat_show(struct seq_file *seq, void *v)
8 {
```

```
9       struct softnet_data *sd = NULL;
10
11      for (int i = 0; i < nr_cpu_ids; i++)
12      {
13          if (cpu_online(i)) {
14              sd = &per_cpu(softnet_data, i);
15              print_netdata(seq, sd);
16          }
17      }
18
19      return 0;
20  }
```

В листинге 3.3 представлен код функции вывода информации о структуре softnet_data. Для softnet_data выводятся: номер cpu; количество обработанных кадров; количество раз, когда у net_rx_action была работа, но бюджета не хватало либо было достигнуто ограничение по времени, прежде чем работа была завершена; количество отброшенных кадров; флаг, находится ли структура в процессе опроса.

Листинг 3.3 – Функция вывода информации о структуре softnet_data

```
1 void print_softnet_data(struct seq_file *seq, struct softnet_data
    *sd){
2     seq_printf(seq, "softnet_data cpu #%d: %d %d %d %d\n", sd->cpu,
        sd->processed, sd->time_squeeze, sd->dropped,
        sd->in_net_rx_action);
3 }
```

В листинге 3.4 представлен код функции вывода информации о backlog очереди. Выводятся: количество раз, когда посредством межпроцессорного прерывания будили CPU для обработки пакетов; длина очереди sk_buff, находящихся в ожидании обработки; длина очереди sk_buff, находящихся в обработке; количество sk_buff, добавленных в очередь.

Листинг 3.4 – Функция вывода информации о backlog очереди

```
1 void print_backlog_data(struct seq_file *seq, struct softnet_data
    *sd) {
```

```
2     seq_printf(seq, "\tbacklog: %d %d %d %d\n", sd->received_rps,
          skb_queue_len(&sd->input_pkt_queue),
          skb_queue_len(&sd->process_queue), sd->input_queue_tail);
3 }
```

В листинге 3.5 представлен код функции вывода информации о napi_struct backlog очереди. Для структуры napi_struct выводятся: состояние; бюджет; номер ядра, на котором была запланирована обработка; длина rx_list, в которую сохраняются не объединённые механизмом GRO кадры; идентификатор структуры. Также выводится количество кадров в GRO–пакетах.

Листинг 3.5 – Функция вывода информации о napi_struct backlog очереди

```
1 void print_backlog_napi_data(struct seq_file *seq, struct
      napi_struct *n){
2     if(n == NULL) return;
3
4     seq_printf(seq, "\tbacklog_napi: %ld %d %d %d %d %ld\n",
          n->state, n->weight, n->list_owner, n->rx_count, n->napi_id);
5     seq_printf(seq, "\tgro_buckets: ");
6     for(int i = 0; i < GRO_HASH_BUCKETS; i++) seq_printf(seq, "%d
          ", n->gro_hash->count);
7     seq_printf(seq, "\n");
8 }
```

В листинге 3.6 представлен Makefiel для компиляции и загрузки модуля.

Листинг 3.6 – Makefiel для компиляции и загрузки модуля

```
1 obj-m += netstat.o
2
3 KDIR ?= /lib/modules/$(shell uname -r)/build
4
5 ccflags-y += -std=gnu18 -Wall
6
7 build:
8     make -C $(KDIR) M=$(shell pwd) modules
9
10 clean:
11     make -C $(KDIR) M=$(shell pwd) clean
```

```
12
13 ins: build
14     sudo insmod netstat.ko
```

4 Исследовательский раздел

4.1 Демонстрация работы программы

На рисунке 4.1 представлена загрузка модуля, обращение к нему через интерфейс proc и выгрузка.

```
→  src git:(main) ✗ sudo insmod netstat.ko
→  src git:(main) ✗ cat /proc/netstat
softnet_data cpu #0: 208076 0 0 0
        backlog: 3199 0 0 170180
        backlog_napi: 0 64 0 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #1: 199848 0 0 0
        backlog: 973 0 0 170344
        backlog_napi: 0 64 1 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #2: 238714 0 0 0
        backlog: 5317 1 0 188424
        backlog_napi: 1 64 2 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #3: 178424 0 0 0
        backlog: 3863 0 0 166202
        backlog_napi: 1 64 3 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #4: 192035 0 0 0
        backlog: 3815 0 0 185099
        backlog_napi: 1 64 4 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #5: 207152 0 0 0
        backlog: 4291 0 0 189332
        backlog_napi: 1 64 5 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #6: 220640 0 0 1
        backlog: 7732 1 0 198233
        backlog_napi: 1 64 6 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
softnet_data cpu #7: 198256 0 0 0
        backlog: 1831 0 0 168812
        backlog_napi: 0 64 7 0 0
        gro_buckets: 0 0 0 0 0 0 0 0
→  src git:(main) ✗ sudo rmmod netstat
→  src git:(main) ✗ sudo dmesg | grep netstat | tail -2
[ 2716.286073] netstat: module loaded
[ 2749.176854] netstat: module unloaded
```

Рисунок 4.1 – Демонстрация работы модуля

4.2 Вывод

В данном разделе были приведён пример работы загружаемого модуля ядра. Разработанная программа выполняет поставленную задачу: выводит информацию о работе сетевой подсистемы.

ЗАКЛЮЧЕНИЕ

Цель, которая была поставлена в начале курсовой работы, была достигнута: разработан загружаемый модуль ядра, предоставляющий пользователю информацию о работе сетевой подсистемы Linux.

Решены все поставленные задачи:

— проведён анализ функций и структур, используемых для обработки сетевых кадров;

— разработан загружаемый модуль ядра, предоставляющий информацию о работе сетевой подсистемы;

— реализован загружаемый модуль ядра.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux Statistics 2024 [Электронный ресурс]. — URL: https://truelist.co/blog/linux-statistics/ (дата обращения: 01.02.2024).

2. Benvenuti C. Understanding Linux Network Internals, 2005. 1062 p.

3. Illustrated Guide to Monitoring and Tuning the Linux Networking Stack: Receiving Data [Электронный ресурс]. — URL: https://blog.packagecloud.io/illustrated-guide-monitoring-tuning-linux-networking-stack\-receiving-data/ (дата обращения: 07.01.2024).

4. softnet_data identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/include/linux/netdevice.h#L3238 (дата обращения: 07.01.2024).

5. Monitoring and Tuning the Linux Networking Stack: Receiving Data [Электронный ресурс]. — URL: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/ (дата обращения: 07.01.2024).

6. ____napi_schedule identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/net/core/dev.c#L4437 (дата обращения: 07.01.2024).

7. napi_struct identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/include/linux/netdevice.h#L354 (дата обращения: 07.01.2024).

8. net_rx_action identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/net/core/dev.c#L6701 (дата обращения: 07.01.2024).

9. Monitoring and Tuning the Linux Networking Stack: Sending Dataa [Электронный ресурс]. — URL: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-sending-data/ (дата обращения: 07.01.2024).

10. __qdisc_run identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/net/sched/sch_generic.c#L410 (дата обращения: 07.01.2024).

11. __netif_reschedule identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/net/core/dev.c#L3087 (дата обращения: 07.01.2024).

12. net_tx_action identifier – Linux source code (v6.7) [Электронный ресурс]. — URL: https://elixir.bootlin.com/linux/v6.7/source/net/core/dev.c#L5123 (дата обращения: 07.01.2024).