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 42 с., 5 рис., 1 табл., 12 источн., 1 прил.
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.

Получение, отправкой и пересылкой трафика занимается сетевая подсистема Linux. Её мониторинг позволят 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:

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

— провести анализ функций и структур, позволяющие получить и вывести информацию о сетевой подсистеме;

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

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:

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

— провести анализ функций и структур, позволяющие получить и вывести информацию о сетевой подсистеме;

— 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 (включая сетевые адаптеры) взаимодействуют с ядром одним из двух способов: опрос и прерывания. Также на практике может применяться комбинация этих методов.

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 приводит к растрате большого количества системных ресурсов и редко применяется.

При использовании **прерываний** устройство генерирует аппаратный сигнал при возникновении определённых событий. Каждое прерывание за-

пускает 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 — наиболее распространённая и стандартная ситуация;

— сбой передачи – драйвер не передаёт это уведомление на более высокие сетевые уровни, так как они узнают о сбое другими способами (тайм-ауты таймера, отрицательные подтверждения и т.д.);

— 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 для хранения кадра максимального размера.

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

Большое количество драйверов обрабатывают сразу несколько кадров при прерывании. Обработчик, зарегистрированный драйвером, загружает кадры и помещает их в очередь ввода ядра, вплоть до максимального количества кадров или конца временного интервала. Ограничение нужно поскольку прерывания отключены, пока запущен обработчик драйвера. Иначе всё процес-

сорное время будет занято лишь обработкой сетевого трафика. Из–за этого у других устройств могут начать переполнится буферы, так как их обработчик не будет своевременно забирать оттуда данные, что приведёт к потерям. 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, в котором код ядра находится 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 протокола более высокого уровня и передать дальше по цепочке обслуживания.

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 позволяет ядру отключать прерывания на гораздо меньшее время:

— устройство генерирует сигнал прерывания;

— процессор выполняет верхнюю половину и блокирует 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.

В сетевом коде используются два типа прерываний NET_TX_SOFTIRQ и NET_RX_SOFTIRQ. Каждый тип softirq может поддерживать массив структур данных типа softnet_data, по одной на процессор, для хранения информации о состоянии текущего softirq и управления их выполнением. 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, сохраняет битовую маску softirq ожидающих для обработки и переходит выполнению обработчика (функция __do_softirq). В моменты обращения к битовой маске (структуре softnet_data), блокируются прерывания. Проходятся по битовой маске в цикле, определяются softirq требующие выполнения и запускаются зарегистрированные обработчики хранящиеся в массиве softirq_vec (для NET_RX_SOFTIRQ это net_rx_action). Их код представлен в листинге 1.1

Листинг 1.1 – Код функций do_softirq и __do_softirq

```
1  asmlinkage __visible void do_softirq(void)
2  {
3      __u32 pending;
4      unsigned long flags;
5
6      if (in_interrupt())
7      return;
8
9      local_irq_save(flags);
10
11     pending = local_softirq_pending();
12
13     if (pending)
14     do_softirq_own_stack();
15
16     local_irq_restore(flags);
17 }
18
19 asmlinkage __visible void __softirq_entry __do_softirq(void)
20 {
21     unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
22     unsigned long old_flags = current->flags;
23     int max_restart = MAX_SOFTIRQ_RESTART;
24     struct softirq_action *h;
25     bool in_hardirq;
26     __u32 pending;
27     int softirq_bit;
```

```
28
29      /*
30      * Mask out PF_MEMALLOC as the current task context is borrowed
            for the
31      * softirq. A softirq handled, such as network RX, might set
            PF_MEMALLOC
32      * again if the socket is related to swapping.
33      */
34      current->flags &= ~PF_MEMALLOC;
35
36      pending = local_softirq_pending();
37
38      softirq_handle_begin();
39      in_hardirq = lockdep_softirq_start();
40      account_softirq_enter(current);
41
42      restart:
43      /* Reset the pending bitmask before enabling irqs */
44      set_softirq_pending(0);
45
46      local_irq_enable();
47
48      h = softirq_vec;
49
50      while ((softirq_bit = ffs(pending))) {
51          unsigned int vec_nr;
52          int prev_count;
53
54          h += softirq_bit - 1;
55
56          vec_nr = h - softirq_vec;
57          prev_count = preempt_count();
58
59          kstat_incr_softirqs_this_cpu(vec_nr);
60
61          trace_softirq_entry(vec_nr);
62          h->action(h);
63          trace_softirq_exit(vec_nr);
64          if (unlikely(prev_count != preempt_count())) {
65              pr_err("huh, entered softirq %u %s %p with
                    preempt_count %08x, exited with %08x?\n",
```

```
66                  vec_nr, softirq_to_name[vec_nr], h->action,
67                  prev_count, preempt_count());
68                  preempt_count_set(prev_count);
69              }
70              h++;
71              pending >>= softirq_bit;
72          }
73
74          if (!IS_ENABLED(CONFIG_PREEMPT_RT) &&
75          __this_cpu_read(ksoftirqd) == current)
76          rcu_softirq_qs();
77
78          local_irq_disable();
79
80          pending = local_softirq_pending();
81          if (pending) {
82              if (time_before(jiffies, end) && !need_resched() &&
83              --max_restart)
84              goto restart;
85
86              wakeup_softirqd();
87          }
88
89          account_softirq_exit(current);
90          lockdep_softirq_end(in_hardirq);
91          softirq_handle_end();
92          current_restore_flags(old_flags, PF_MEMALLOC);
93      }
```

В сетевой подсистеме NET_RX_SOFTIRQ используется для обработки входящего трафика, а NET_TX_SOFTIRQ исходящего. Их обработчики регистрируются при инициализации устройства. Они имеют приоритет ниже чем HI_SOFTIRQ, но выше чем у TASKLET_SOFTIRQ. Такая расстановка приоритетов гарантирует, что другие высокоприоритетные задачи могут выполняться оперативно и своевременно, даже когда система находится под высокой сетевой нагрузкой.

Каждый процессор имеет свою собственную структуру данных для управления входящим и исходящим трафиком. Это структура softnet_data, которая представлена в листинге 1.2.

Листинг 1.2 – Код структуры softnet_data

```
1  struct softnet_data {
2      struct list_head    poll_list;
3      struct sk_buff_head process_queue;
4
5      /* stats */
6      unsigned int        processed;
7      unsigned int        time_squeeze;
8      #ifdef CONFIG_RPS
9      struct softnet_data *rps_ipi_list;
10     #endif
11
12     bool            in_net_rx_action;
13     bool            in_napi_threaded_poll;
14
15     #ifdef CONFIG_NET_FLOW_LIMIT
16     struct sd_flow_limit __rcu *flow_limit;
17     #endif
18     struct Qdisc        *output_queue;
19     struct Qdisc        **output_queue_tailp;
20     struct sk_buff      *completion_queue;
21     #ifdef CONFIG_XFRM_OFFLOAD
22     struct sk_buff_head xfrm_backlog;
23     #endif
24     /* written and read only by owning cpu: */
25     struct {
26         u16 recursion;
27         u8  more;
28         #ifdef CONFIG_NET_EGRESS
29         u8  skip_txqueue;
30         #endif
31     } xmit;
32     #ifdef CONFIG_RPS
33     /* input_queue_head should be written by cpu owning this struct,
34     * and only read by other cpus. Worth using a cache line.
35     */
36     unsigned int        input_queue_head
           ____cacheline_aligned_in_smp;
37
38     /* Elements below can be accessed between CPUs for RPS/RFS */
39     call_single_data_t  csd ____cacheline_aligned_in_smp;
```

```
40      struct softnet_data *rps_ipi_next;
41      unsigned int        cpu;
42      unsigned int        input_queue_tail;
43      #endif
44      unsigned int        received_rps;
45      unsigned int        dropped;
46      struct sk_buff_head input_pkt_queue;
47      struct napi_struct  backlog;
48
49      /* Another possibly contended cache line */
50      spinlock_t      defer_lock ____cacheline_aligned_in_smp;
51      int         defer_count;
52      int         defer_ipi_scheduled;
53      struct sk_buff      *defer_list;
54      call_single_data_t  defer_csd;
55  };
```

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 с входными кадрами, ожидающими обработки;

— 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 — список устройств, которым есть что передать;

— 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 по причине нехватки места в очереди обработки;

— 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. Но всё же NAPI не может совсем избавить нас от прерываний. Он позволяет драйверу устройства регистрировать функцию poll, вызываемую подсистемой NAPI для сбора фрейма данных.

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:

— 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, но изначально тот находится в неактивном состоянии;

— прибывает пакет, и сетевая карта напрямую отправляет его в память;

— сетевая карта генерирует IRQ посредством запуска обработчика прерываний в драйвере

— драйвер 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 фреймов данных позволил уменьшить нагрузку по сравнению со старым методом, поскольку многие фреймы могут одновременно приниматься без необходимости одновременного 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 пакет от прибытия до приёмного буфера сокета выглядит так:

— драйвер загружается и инициализируется;

— пакет прибывает из сети в сетевую карту;

— пакет копируется (посредством DMA) в кольцевой буфер памяти ядра;

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

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

— на каждом CPU системы работают процессы ksoftirqd. Они регистрируются во время загрузки. Эти процессы вытаскивают пакеты из кольцевого буфера с помощью вызова NAPI-функции poll, зарегистрированной драйвером устройства во время инициализации;

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

— данные, отправленные напрямую в память (DMA), передаются для дальнейшей обработки на сетевой уровень в виде 'skb';

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

— фреймы сетевых данных передаются из очереди на уровни протоколов;

— уровни протоколов обрабатывают данные;

— данные добавляются в буферы приёма, прикреплённые к сокетам уровнями протоколов.

При получении кадра на сетевой карте генерируется прерывание, обработчик которого был зарегистрирован при инициализации драйвера. В самом обработчике выполняется выполняется какой–то код драйвера и вызывается функция napi_schedule (обёртка для ____napi_schedule), в которую как параметр предаётся napi_struct драйвера. Её код представлен в листинге 1.3.

Листинг 1.3 – Код функции ____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           /* Paired with smp_mb__before_atomic() in
10          * napi_enable()/dev_set_threaded().
11          * Use READ_ONCE() to guarantee a complete
12          * read on napi->thread. Only call
13          * wake_up_process() when it's not NULL.
14          */
15          thread = READ_ONCE(napi->thread);
16          if (thread) {
17              /* Avoid doing set_bit() if the thread is in
18              * INTERRUPTIBLE state, cause napi_thread_wait()
19              * makes sure to proceed with napi polling
20              * if the thread is explicitly woken from here.
21              */
22              if (READ_ONCE(thread->__state) != TASK_INTERRUPTIBLE)
23              set_bit(NAPI_STATE_SCHED_THREADED, &napi->state);
24              wake_up_process(thread);
25              return;
26          }
27      }
28
29      list_add_tail(&napi->poll_list, &sd->poll_list);
30      WRITE_ONCE(napi->list_owner, smp_processor_id());
31      /* If not called from net_rx_action()
32      * we have to raise NET_RX_SOFTIRQ.
33      */
34      if (!sd->in_net_rx_action)
35      __raise_softirq_irqoff(NET_RX_SOFTIRQ);
36  }
```

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 1.4, 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 1.5.

Листинг 1.4 – Код структуры napi_struct

```
1  struct napi_struct {
2      /* The poll_list must only be managed by the entity which
3      * changes the state of the NAPI_STATE_SCHED bit.  This means
4      * whoever atomically sets that bit can add this napi_struct
5      * to the per-CPU poll_list, and whoever clears that bit
6      * can remove from the list right before clearing the bit.
7      */
8      struct list_head    poll_list;
9
10     unsigned long       state;
11     int         weight;
12     int         defer_hard_irqs_count;
13     unsigned long       gro_bitmask;
14     int         (*poll)(struct napi_struct *, int);
15     #ifdef CONFIG_NETPOLL
16     /* CPU actively polling if netpoll is configured */
17     int         poll_owner;
18     #endif
19     /* CPU on which NAPI has been scheduled for processing */
20     int         list_owner;
21     struct net_device   *dev;
22     struct gro_list     gro_hash[GRO_HASH_BUCKETS];
23     struct sk_buff      *skb;
24     struct list_head    rx_list; /* Pending GRO_NORMAL skbs */
25     int         rx_count; /* length of rx_list */
26     unsigned int        napi_id;
27     struct hrtimer      timer;
28     struct task_struct  *thread;
29     /* control-path-only fields follow */
30     struct list_head    dev_list;
31     struct hlist_node   napi_hash_node;
32 };
```

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.

Листинг 1.5 – Код функции net_rx_action

```
1  static __latent_entropy void net_rx_action(struct softirq_action *h)
2  {
3      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                 /* We need to check if ____napi_schedule()
26                 * had refilled poll_list while
27                 * sd->in_net_rx_action was true.
28                 */
29                 if (!list_empty(&sd->poll_list))
30                 goto start;
31                 if (!sd_has_rps_ipi_waiting(sd))
32                 goto end;
33             }
34             break;
35         }
36
37         n = list_first_entry(&list, struct napi_struct, poll_list);
38         budget -= napi_poll(n, &repoll);
39
40         /* If softirq window is exhausted then punt.
```

```
41          * Allow this to run for 2 jiffies since which will allow
42          * an average latency of 1.5/HZ.
43          */
44          if (unlikely(budget <= 0 ||
45          time_after_eq(jiffies, time_limit))) {
46              sd->time_squeeze++;
47              break;
48          }
49      }
50
51      local_irq_disable();
52
53      list_splice_tail_init(&sd->poll_list, &list);
54      list_splice_tail(&repoll, &list);
55      list_splice(&list, &sd->poll_list);
56      if (!list_empty(&sd->poll_list))
57      __raise_softirq_irqoff(NET_RX_SOFTIRQ);
58      else
59      sd->in_net_rx_action = false;
60
61      net_rps_action_and_irq_enable(sd);
62      end:;
63  }
```

Функция итерирует по списку структур NAPI, стоящих в очереди текущего CPU, поочерёдно извлекает каждую структуру работает с ней. Цикл обработки ограничивает объём работы и время исполнения зарегистрированных NAPI-функций poll. Он делает это двумя способами: отслеживая рабочий бюджет и проверяет затраченное время. Таким образом ядро не позволяет обработке пакетов занять все ресурсы CPU. budget — это весь доступный бюджет, который будет разделён на все доступные NAPI-структуры, зарегистрированные на этот CPU. Бюджет является настраиваемой величиной, но функция всё ещё будет иметь ограничение по времени.

Выбрав NAPI–структуру (napi_struct) вызывается функция poll, которая возвращает количество обработанных фреймов. Сама функция функция собирает сетевые данные и отправляет их в стек для дальнейшей обработки. Затем этот 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. Иначе она должна отключить NAPI. NAPI будет снова включён при получении следующего IRQ.

Выход из цикла net_rx_action будет совершён, если: список poll, зарегистрированный для данного CPU, больше не содержит NAPI–структур, остаток бюджета <= 0, 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. сли 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» пакетов. Это позволяет снизить нагрузку на CPU. Пусть передаётся большой файл, и большинство пакетов содержат чанки данных из этого файла. Вместо отправки по стеку маленьких пакетов по одному, входящие пакеты можно комбинировать в один большой. А затем уже передавать его по стеку. Таким образом уровни протоколов обрабатывают заголовки одного пакета, при этом передавая пользовательской программе более крупные чанки. Но этой оптимизации присуща проблема потери информации. Если какой-то пакет имеет настроенную важную опцию или флаг, то эта опция или флаг могут быть потеряны при объединении с другими пакетами.

Функция napi_gro_receive, вызываемая в poll функции драйверов, 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 было 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 структуры данных, 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, код которой представлен в листинге 1.6.

Листинг 1.6 – Код функции enqueue_to_backlog

```
1  static int enqueue_to_backlog(struct sk_buff *skb, int cpu,
2  unsigned int *qtail)
3  {
4      enum skb_drop_reason reason;
5      struct softnet_data *sd;
6      unsigned long flags;
7      unsigned int qlen;
8
9      reason = SKB_DROP_REASON_NOT_SPECIFIED;
10     sd = &per_cpu(softnet_data, cpu);
11
12     rps_lock_irqsave(sd, &flags);
13     if (!netif_running(skb->dev))
14     goto drop;
15     qlen = skb_queue_len(&sd->input_pkt_queue);
16     if (qlen <= READ_ONCE(netdev_max_backlog) &&
           !skb_flow_limit(skb, qlen)) {
```

```
17          if (qlen) {
18              enqueue:
19              __skb_queue_tail(&sd->input_pkt_queue, skb);
20              input_queue_tail_incr_save(sd, qtail);
21              rps_unlock_irq_restore(sd, &flags);
22              return NET_RX_SUCCESS;
23          }
24
25          /* Schedule NAPI for backlog device
26          * We can use non atomic operation since we own the queue
                lock
27          */
28          if (!__test_and_set_bit(NAPI_STATE_SCHED,
              &sd->backlog.state))
29              napi_schedule_rps(sd);
30              goto enqueue;
31      }
32      reason = SKB_DROP_REASON_CPU_BACKLOG;
33
34      drop:
35      sd->dropped++;
36      rps_unlock_irq_restore(sd, &flags);
37
38      dev_core_stats_rx_dropped_inc(skb->dev);
39      kfree_skb_reason(skb, reason);
40      return NET_RX_DROP;
41  }
```

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 при-

меняется weight. Структура NAPI предоставляется в ходе инициализации сетевой подсистемы. 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. process_backlog соблюдает тот же договор с NAPI, что и драйверы устройства: NAPI отключается, если не расходуется весь вес. Поллер перезапускается посредством вызова ____napi_schedule из enqueue_to_backlog.

ЗАКЛЮЧЕНИЕ

Цель, которая была поставлена в начале научно–исследовательской работы, была достигнута: проведён обзор и сравнение существующих методов моделирования многофункциональных центров обслуживания.

Решены все поставленные задачи:

— изучены основные понятия моделирования многофункциональных центров обслуживания;

— описаны и классифицированы существующие методы;

— произведён сравнительный анализ рассмотренных методов.

В ходе исследования были определены особенности, преимущества и недостатки рассмотренных методов. В итоге был сделан вывод о том, что лучше всего для моделирования многофункциональных центров обслуживания подходят сети Петри, так как они позволяют моделировать стохастические и параллельные системы, а также концентрируются на локальных событиях, что позволяет получить подробную информацию о работе всех элементов системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

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

2. Градов В.М. Компьютерное моделирование / В.М. Градов, Г.В. Овечкин, П.В. Овечкин, И.В. Рудаков — М.:КУРС ИНФРА–М, 2019. — 264 С.

3. Посещаемость «Мои документы» в 2022 году [Электронный ресурс]. — URL: `https://www.mos.ru/news/item/117681073/` (дата обращения: 12.11.2023).

4. Расширение концепции ООО–модели для систем массового обслуживания на примере многофункционального центра предоставления государственных и муниципальных услуг / А.В. Чуев, С.А. Юдицкий, В.З. Магергут // Экономика. Информатика. — 2015. — №. 1. – С. 85–93.

5. Пронникова Т.Ю. Применение имитационного моделирования для оптимизации бизнес-процессов обслуживания клиентов в многофункциональном центре / Т.Ю. Пронникова, М.Н. Рассказова // Прикладная математика и фундаментальная информатика. — 2022. — С. 122-123.

6. Сутягина Н. И. Моделирование деятельности многофункционального центра как системы массового обслуживания // Карельский научный журнал. — 2015. — №. 1. — С. 199–203.

7. Моделирование систем / С.П. Бобков, Д.О. Бытев // —Иваново:Изд. ИвГХТУ, 2008. — 156 с.

8. Теория автоматов / Ожиганов А.А. // — СПб.:НИУ ИТМО, 2013. — 84 с.

9. Моделирование систем / Альсова О.К. // — Новосибирск:Изд-во НГТУ, 2007. — 72 с.

10. Блюмин, С.Л. Дискретное моделирование систем автоматизации и управления / С.Л. Блюмин, А.М. Корнеев. — Липецк:ЛЭГИ, 2005. — 124 с.

11. Осипов Г.С. Математическое и имитационное моделирование систем массового обслуживания / Г.С. Осипов — М.: Издательский дом Академии Естествознания, 2017. — 56 с.

12. Григорьева Т. Е., Донецкая А. А., Истигечева Е. В. Моделирование одноканальных и многоканальных систем массового обслуживания на примере билетной кассы автовокзала / Т.Е. Григорьева, А.А. Донецкая, Е.В. Истигечева // Вестник Воронежского института высоких технологий. – 2017. — №. 1. — С. 35–38.

13. Мальков М.В. Сети Петри и моделирование / М.В. Мальков, С.Н. Малыгина // Труды Кольского научного центра РАН. — 2010. — №. 3. — С. 35–40.

ПРИЛОЖЕНИЕ А

Презентация научно-исследовательской работы

Презентация научно-исследовательской работы содержит 14 слайдов, на которых представлено краткое описание научно-исследовательской работы.