ФГАОУ ВО «Национальный исследовательский университет «Московский институт электронной техники»

Направление: Технические дисциплины

Выпускная квалификационная работа

**Разработка программного модуля мониторинга и корректировки прерываний сетевых интерфейсов**

**Работу выполнил:**

Поселенов Кирилл Владимирович  
Студент 4 курса

ФГАОУ ВО Национальный исследовательский университет «Московский институт электронной техники»

**Научный руководитель:**

Федотова Елена Леонидовна

К.п.н., доцент

ФГАОУ ВО Национальный исследовательский университет «Московский институт электронной техники»

Москва, 2021

# АННОТАЦИЯ

Актуальность данной темы заключается в том, что разрабатываемый модуль упростит корректировку масок для оператора ЭВМ, также программный модуль уменьшит количество ошибок, связанных с человеческим фактором.

Объектом исследования являются прерывания, генерируемые сетевой карты. Предметом исследования является корректировка масок прерываний сетевых интерфейсов.

Целью данной работы является снижение влияния событий сетевых интерфейсов на формирование и корректировку масок прерываний.

В рамках данной работы исследуется предметная область, решаются задачи по сравнительному анализу существующих аналогичных программных решений, выбору языка и среды программирования для реализации ПМ МСИ, разработке схем данных и алгоритма. Также работа описывает процесс разработки, тестирования и отладки.

В данной работе применен метод корректировки прерываний путем изменения масок прерываний в циклическом режиме.

В результате работы был разработан программный модуль в виде программы-демона, которая выставляет маски прерываний в соответствующие файлы smp\_affinity. Также была достигнута цель работы, а именно удалось отказаться от использования событий сетевых интерфейсов.

Данный модуль способен равномерно распределять нагрузку обработки сетевых пакетов по доступным ядрам для заданных интерфейсов.

# СОДЕРЖАНИЕ

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ 5](#_Toc75294297)

[ВВЕДЕНИЕ 6](#_Toc75294298)

[1. ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ 8](#_Toc75294299)

[1.1. Исследование предметной области 8](#_Toc75294300)

[1.2. Обзор существующих программных решений 11](#_Toc75294301)

[1.3. Концептуальная модель 20](#_Toc75294302)

[1.4. Схема данных программного модуля 22](#_Toc75294303)

[Выводы по исследовательскому разделу 26](#_Toc75294304)

[2. КОНСТРУКТОРСКИЙ РАЗДЕЛ 28](#_Toc75294305)

[2.1. Схема алгоритма 28](#_Toc75294306)

[2.2. Анализ языков программирования 32](#_Toc75294307)

[2.3. Анализ сред разработки 34](#_Toc75294308)

[2.4. Анализ кода ПМ МСИ 36](#_Toc75294309)

[Выводы по конструкторскому разделу 48](#_Toc75294310)

[3. ТЕХНОЛОГИЧЕСКИЙ РАЗДЕЛ 50](#_Toc75294311)

[3.1. Сборка программ на языке Си 50](#_Toc75294312)

[3.2. Отладка ПМ МСИ 51](#_Toc75294313)

[3.3. Анализ средств тестирования 59](#_Toc75294314)

[3.4. Тестирование ПМ МСИ методом черного ящика 61](#_Toc75294315)

[Выводы по технологическому разделу 71](#_Toc75294316)

[ЗАКЛЮЧЕНИЕ 73](#_Toc75294317)

[СПИСОК ЛИТЕРАТУРЫ 74](#_Toc75294318)

[ТЕХНИЧЕСКОЕ ЗАДАНИЕ 78](#_Toc75294319)

[РУКОВОДСТВО ОПЕРАТОРА 85](#_Toc75294320)

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

ОС – операционная система;

ООП – объектно-ориентированное программирование;

ЭВМ – электронно-вычислительная машина;

ЦП – центральный процессор;

ЯП – язык программирования;

NAPI – new application programming interface;

ПО – программное обеспечение;

GPL - general public license;

IRQ – interrupt request;

CPU – central processing unit;

PID - process Identifier;

IPsec – IP Security;

GDB – GNU Debbuger;

NUMA – none – uniform memory access;

RX – receive;

TX – transmit;

ПМ МСИ – программный модуль мониторинга сетевых интерфейсов.

# ВВЕДЕНИЕ

В современных IT - компаниях, которые связанны с обработкой данных через сеть Интернет, устанавливаются специальные ЭВМ. На такие машины в каждый момент времени поступает огромное количество информации, которая представлена в виде сетевых пакетов, поэтому скорость обработки этих пакетов играет огромную роль.

За обработку сетевых пакетов в ЭВМ отвечает отдельное устройство – сетевая карта. Сетевая карта – это аппаратный (внешний или встроенный в материнскую плату) компонент, с помощью которого компьютер подключается к сети (локальной или Интернет). Сетевая карта работает посредником между компьютером и сетью передачи данных.

Все пакеты, отправляющиеся и принимающиеся сетевой картой, проходят через сетевые интерфейсы. Сетевой интерфейс – это программная реализация интерфейса для сетевой карты, с помощью которой оператор может контролировать прием пакетов.

В ОС Linux у каждого сетевого интерфейса имеется собственная маска прерывания, которая отвечает за то, какие ядра процессора будут обрабатывать пакеты, поступающие для этого интерфейса. Прерывание – это автоматическое изменение хода выполнения программы (запуск подпрограммы обработчика прерывания), вызванное событием (программным или аппаратным) или совокупностью событий, возникших в некоторой части вычислительной системы. Для того чтобы система могла быстро обрабатывать пакеты данных, оператору необходимо грамотно распределить работу по всем доступным ядрам процессора, потому что для специализированных ЭВМ крайне важно обрабатывать пакеты за минимальное время, чтобы быстродействие машины сохранялось на высоком уровне.

Ввиду того, что сложность процессоров постоянно растет, количество ядер увеличивается, грамотное распределение работы по ядрам процессора становится проблематичным. Также в ЭВМ может существовать большое количество различных сетевых интерфейсов, для каждого из которых необходимо задать уникальную маску.

Актуальность данной темы заключается в том, что разрабатываемый модуль упростит корректировку масок для оператора ЭВМ, также программный модуль уменьшит количество ошибок, связанных с человеческим фактором.

В рамках данной работы исследуется предметная область, решаются задачи по сравнительному анализу существующих аналогичных программных решений, выбору языка и среды программирования для реализации ПМ МСИ, разработке схем данных и алгоритма. Также работа описывает процесс разработки, тестирования и отладки.

Выполнение выпускной квалификационной работы проходило в ООО «С-Терра СиЭсПи», специализирующейся на разработке и производстве средств сетевой информационной безопасности, а также ПО для реализации этих средств.

Разрабатываемый ПМ МСИ имеет практическую значимость для решения задач мониторинга и корректировки сетевых интерфейсов.

Целью данной работы является повышение точности корректировки масок прерываний сетевых интерфейсов.

Задачи:

* исследование предметной области;
* сравнительный анализ существующих аналогов;
* выбор языка и среды программирования;
* разработка схемы данных ПМ МСИ;
* разработка схем алгоритмов ПМ МСИ;
* разработка пользовательского интерфейса;
* программная реализация;
* отладка и тестирование ПМ МСИ;
* разработка руководства оператора.

Пояснительная записка состоит из введения, трех разделов, заключения, списка литературы, приложений и занимает 75 страницы.

В исследовательском разделе изучена предметная область, осуществлен обзор существующих программных решений, разработана схема данных.

В конструкторском разделе приведена схема алгоритма сервиса и подпрограммы, произведен анализ языков и сред программирования, разобраны ключевые детали кода программы.

В технологическом разделе представлен процесс отладки с помощью GDB и Valgrind, а также проведено тестирование ПМ МСИ.

# ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ

## Исследование предметной области

Залогом эффективной работы ЭВМ, которые установлены для обработки большого потока данных через сеть Интернет, является равномерное использование ядер процессора в обработке прерываний от сетевых интерфейсов. Если оператор ЭВМ будет нерационально использовать доступные ядра процессора, это приведет к снижению пропускной способности данной ЭВМ, вследствие чего увеличится время обработки одного сетевого пакета.

Существует два основных механизма обработки сетевых пакетов в ОС Linux:

1. Традиционная модель (модель прерываний).
2. NAPI.

Суть традиционной модели заключается в том, что каждый приходящий пакет на вход сетевой карты порождает прерывание для ЦП по IRQ – линии, тем самым уведомляя систему о том, что поступил новый пакет. Последовательность действий при появлении пакета отражена на рисунке:



Рис. 1.1. Схема последовательности событий при появлении нового сетевого пакета

Одна из проблем данной концепции состоит в том, что при большом потоке данных, такой тип обработки значительно снижает быстродействие системы. Все больше времени уходит на обработку прерываний от сетевой карты, что не позволяет корректно работать операционной системе, что, в свою очередь, нарушает работу пользовательских программ. Такая концепция подходит лишь для низкоскоростных систем, где число пакетов в секунду будет около 5 – 10 тысяч.

Для того чтобы снизить количество прерываний, которые возникают при появлении пакета разработана концепция NAPI. Суть данной концепции состоит в том, чтобы не обрабатывать пакеты по отдельности, а обработать сразу несколько пакетов. При поступлении первого пакета генерируется прерывание по определенному номеру IRQ. После этого включается подсистема NAPI, драйвер отключает прерывания от сетевой карты и начинает обрабатывать последующие входные пакеты. Этот метод называется методом опроса. Данная концепция позволяет сэкономить ресурсы ЦП при высокоскоростном трафике, так как обработка прерываний высоко затратное действие.

Так как на предприятии через ЭВМ проходит большой объем данных, выбрана подсистема NAPI.

Также необходимо ввести понятие NUMA и NUMA NODE. NUMA – неравномерный доступ CPU к памяти другого CPU. NUMA позволяет разместить CPU таким образом, чтобы CPU получали доступ к оперативной памяти максимально быстро, чтобы путь от CPU до оперативной памяти был минимальным. При этом CPU не запрещается обращаться к «чужим» участкам памяти, которые принадлежат другим CPU.

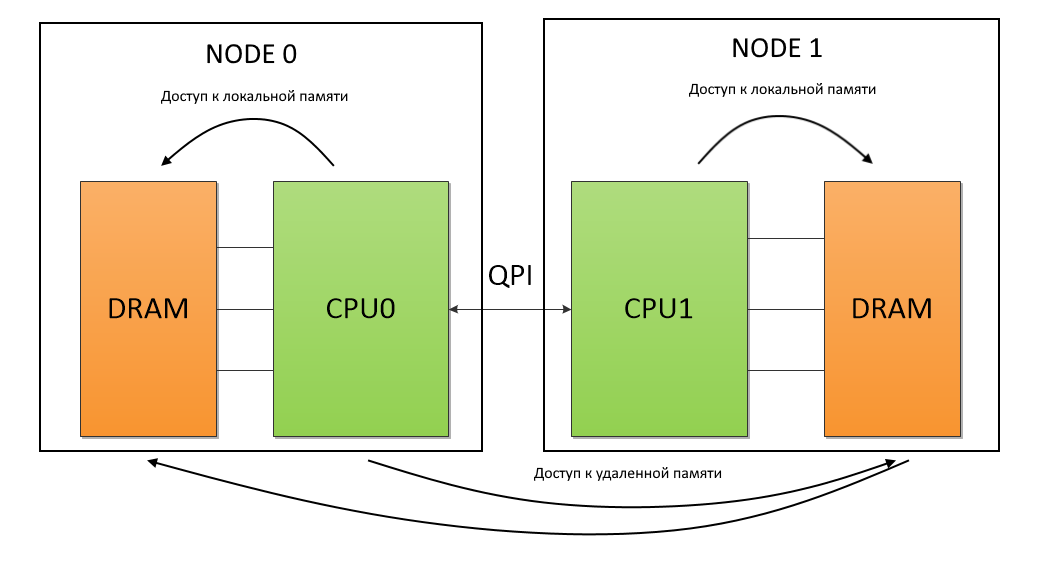


Рис. 1.2. Схематичное изображение узлов NUMA и связи между ними

Объединение CPU и участка оперативной памяти называют NUMA NODE.

Для того чтобы правильно настроить работу сетевой карты ЭВМ, оператор должен задать маску прерывания для сетевого интерфейса, таким образом, чтобы было задействовано как можно больше доступных ядер. Это снизит время обработки пакетов, так как они будут обрабатываться параллельно друг другу. NUMA NODE называется узел, в котором объеден CPU и участок оперативной памяти для этого CPU.

Вся информация о прерываниях того или иного интерфейса хранится в специальных файлах – smp\_affinity. В этом файле оператор может узнать, какая маска установлена для конкретного прерывания. Маска записывается в виде 16-тиричного числа. Например, для того чтобы назначить для обработки прерывания 32 ядра из 64 доступных ядер маска будет выглядеть следующим образом: ffffffff,00000000.

После ручной установки маски прерывания, оператор не может быть уверенным, что данная маска не измениться спустя какое-то время, так как подсистемы ОС Linux могут переопределять параметры работы сетевой карты. В случае если оператору необходимо будет сконфигурировать работу большого числа интерфейсов, будет затрачено значительное количество времени, также это влечет за собой и увеличение вероятности возникновения ошибок, связанных с человеческим фактором.

Для того чтобы автоматизировать данный процесс, внутри компании создано решение, которое выполнено в виде драйвера, работающего на уровне ядра.

Актуальность данной работы заключается в том, что существующее решение стало отрабатывать некорректно. Проблема состоит в том, что через драйвер нет надёжного способа выставить нужные оператору параметры работы сетевых карточек. Сейчас в драйвере регистрируются обработчики событий NETDEV\_UP/NETDEV\_DOWN/NETDEV\_CHANGE от сетевых интерфейсов. Но моменты, когда вызываются эти обработчики, могут быть не всегда подходящими.

Например, драйвер сетевой карточки может динамически изменить параметры её работы – в том числе, изменить привязку RX/TX очередей сетевой карты к номерам CPU, которые должны обрабатывать прерывания от этих очередей, и не вызвать обработчик NETDEV\_CHANGE.

Также, другие подсистемы ядра Linux могут переопределить параметры работы сетевых карточек - это может зависеть от порядка регистрации обработчиков событий от сетевых интерфейсов драйвером, реализованным в компании.

В драйвере происходит парсинг файлов виртуальной файловой системы /proc (например, /proc/interrupts), которые создаёт ядро для пользовательских процессов ОС - для подсистем ядра (и драйверов), должны использоваться другие механизмы получения той же информации, что в этих файлах.

Также, в драйвере выполняется запись в файлы /proc (например, /proc/irq/<irq\_num>/smp\_affinity). Эти файлы ядро также создаёт для взаимодействия с пользовательскими процессами ОС, и писать в эти файлы из ядра выглядит не совсем корректным.

Поэтому принято решение о создании отдельного сервиса для мониторинга и корректировки прерываний сетевых интерфейсов через файлы /proc на уровне пользователя.

## Обзор существующих программных решений

Перед разработкой программного модуля произведен анализ существующих решений. В качестве таких решений рассмотрены:

* команды для терминала, которые оператор будет вводить вручную для каждого из необходимых интерфейсов;
* драйвер С-Терры, текущее решение, которое используют на предприятии;
* драйвер igb, разработанный компанией Intel;
* демон irqbalance.

На предприятии сформированы требования, которым должно соответствовать решение.

Результаты данного анализа приведены в таблице 1.1.

Табл. 1.1.

Сравнительный анализ готовых программных решений

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Программные  решения  Критерии | Команды для терминала [1] | Драйвер С-Терры (текущее решение) [2] | Драйвер igb [3] | Демон irqbalance [4] |
| Возможность записи в файл smp\_affinity | Да | Да | Нет | Да |

Продолжение табл. 1.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Возможность работы в фоновом режиме | Нет | Работает в режиме ядра | Работает в режиме ядра | Да |
| Просмотр промежуточных результатов | Нет | Да | Да (только в режиме отладки) | Да |
| Возможность повторной работы модуля без участия оператора | Нет | Нет | Нет | Да |
| Работа программы на уровне пользователя | Да | Нет | Нет | Да |
| Возможность работы в интерактивном режиме | Да | Нет | Нет | Да |
| Возможность чтения информации о прерываниях из специального файла | Нет | Нет | Нет | Нет |
| Наличие «сухого» режима | Нет | Нет | Нет | Нет |

Источники информации:

[1] https://www.kernel.org/doc/Documentation/IRQ-affinity.txt

[2] https://www.s-terra.ru/

[3] https://www.intel.ru/content/www/ru/ru/support/articles/000005480/network-and-io/ethernet-products.html

[4] https://github.com/Irqbalance/irqbalance

Подробный обзор достоинств и недостатков каждого из представленных решений приведен ниже.

#### Команды для терминала

В ОС Linux большой объем работы происходит в терминале. Терминал предназначен для взаимодействия пользователя с системой по средствам специальных команд. В данном случае с помощью терминала оператор может сконфигурировать работу сетевой карты компьютера, настроить маски прерывания для сетевых интерфейсов.

Пример использования этих команд. Для того чтобы узнать номер IRQ нужного оператору сетевого интерфейса необходимо ввести команду «cat /proc/interrupts»

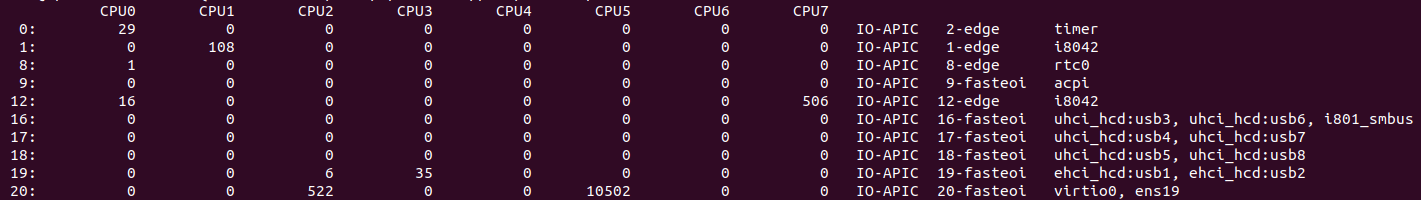


Рис. 1.3. Результат выполнения команды «cat /proc/interrupts»

Также данная команда позволяет узнать, сколько каждое ядро обработало прерываний. В данном случае рассматривается прерывание под номером 20 сетевого интерфейса ens19. В текущий момент его обрабатывает ядро под номером 2 и 5 (CPU2 и CPU5). Для того чтобы узнать как выглядит маска в данный момент, воспользуемся командной «cat /proc/irq/20/smp\_affinity».



Рис. 1.4. Результат выполнения команды «cat /proc/irq/20/smp\_affinity»

Получено число, которое является маской для данного прерывания. Для того чтобы изменить эту маску необходимо ввести команду «echo 2 > /proc/irq/20/smp\_affinity» для того чтобы прерывание под номером 20 обрабатывало только CPU1. Проверить это можно, выполнив команду «cat /proc/interrupts».

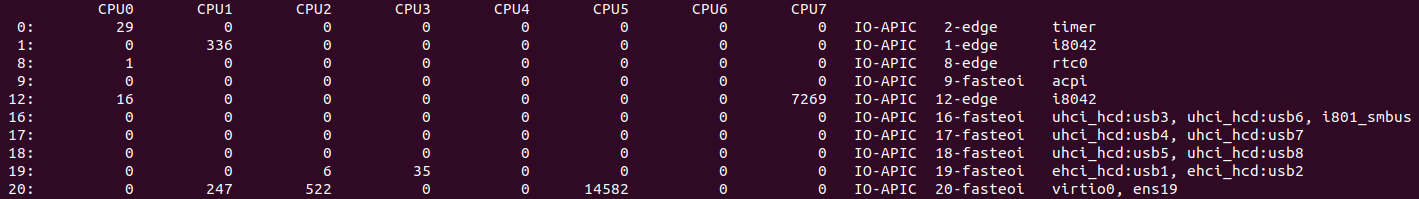


Рис. 1.5. Результат выполнения команды «cat /proc/interrupts» после изменения маски

Как видно из рисунка, прерывание под номером 20 от сетевого интерфейса ens19 теперь обрабатывает только ядро CPU1 (так как число пакетов увеличивается только у CPU1). Для того чтобы убедиться, что теперь прерывание под номером 20 действительно обрабатывает CPU1 необходимо выполнить команду «cat /proc/interrupts» спустя некоторое время. Доказательством, что данное прерывание обрабатывает только CPU1 будет то, что в число прерываний для IRQ 20 увеличивается только для CPU1.

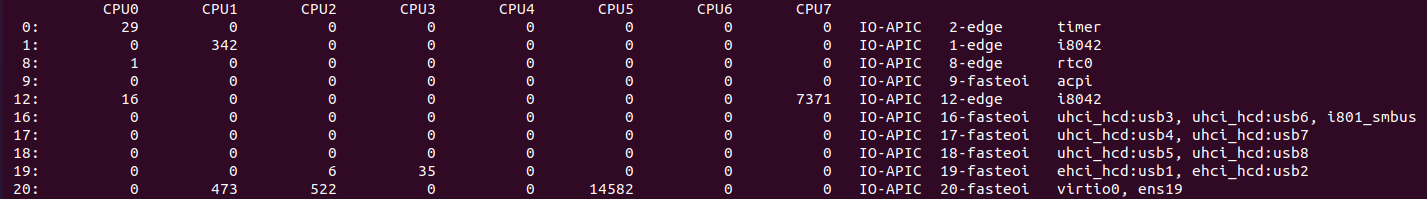


Рис. 1.6. Результат выполнения команды «cat /proc/interrupts» после изменения маски спустя время

Сравнивая результаты, полученные после выполнения команды «cat /proc/interrupts» до и после изменения маски, можно заметить, что теперь прерывание под номером 20 обрабатывается только CPU1.

Таким образом, оператор может вручную изменять маски прерываний сетевых интерфейсов и тем самым конфигурировать работу системы.

Проведен анализ достоинств и недостатков данного решения.

* достоинства:
  + простота использования. Для того чтобы изменить какую-либо маску у конкретного интерфейса нужно написать всего одну команду (пример: для того чтобы задать работу 32 последних ядер из 64 возможных для прерывания под IRQ номером 20 необходимо написать данную команду: # echo 0xffffffff,00000000 > /proc/irq/20/smp\_affinity);
  + доступность информации для оператора. В сети Интернет существует большое количество различных источников, в которых подробно расписана работа с такими командами;
  + минимальное использование ресурсов ОС. Для того чтобы изменить маску необходимо только запустить терминал и ввести команду, не нужно запускать дополнительных приложений, которые будут расходовать ресурсы ОС;
  + привязка конкретных CPU. При формировании маски указывается 16-тиричное число, которое отражает номера конкретных CPU. Поэтому оператор может быть уверен, что после установки маски прерывания будут обрабатывать именно те CPU, которые он указал в маске.
* недостатки:
  + ручная работа. Оператору приходится все проделывать вручную. Для каждого из заданных интерфейсов необходимо вычислить маску, узнать номер прерывания IRQ, соответствующие данному интерфейсу, установить маску по этому номеру IRQ. Число конфигурируемых интерфейсов может быть велико, поэтому на это будет потрачено большое количество времени. Так как это ручная работа живого человека, то повышается вероятность возникновения ошибок;
  + изменение масок прерывания. Если возникает потребность изменить маски прерывания, то оператору вновь придется проделывать большое количество работы вручную;
  + невозможность работы без участия оператора. Все эти команды необходимо прописывать оператору;
  + нет гарантии, что с течением времени маска не изменится. Существуют подсистемы ОС Linux, которые способны переопределить параметры работы сетевых карточек. Поэтому оператору необходимо будет периодически проверять маску у каждого интерфейса;
  + для того чтобы изменить маску прерывания, оператору необходимо иметь доступ к пользователю root. Без доступа к пользователю root оператор не сможет изменить маску прерывания.
* вывод по решению: данное решение может подойти для разовых точечных корректировок, а не для постоянного мониторинга за сетевыми интерфейсами.

#### Драйвер С-Терры vpnirq driver

Проведен анализ текущего решения, которое на данный момент используется внутри компании. Данное решение выполнено в виде драйвера, которое работает в режиме ядра.

* достоинства:
  + задание имен сетевых интерфейсов в конфигурационном файле. Вся информация о сетевых интерфейсах указывается в конфигурационном файле. Данная возможность позволяет оператору не узнавать номера IRQ для всех интерфейсов, так как драйвер самостоятельно вычисляет IRQ номер прерывания по названию сетевого интерфейса (например, eth0:3/4);
  + задание количества CPU, которые будут задействованы в обработке прерывания в удобном виде. В конфигурационном файле помимо имени интерфейса указывается и количество ядер;
  + возможность записи в файлы spm\_affinity. В драйвере производится запись сформированной маски прерывания в файл smp\_affinity по заданному IRQ;
  + возможность автономной работы. Необходимо лишь один раз сформировать конфигурационный файл и вшить драйвер в ОС, затем драйвер будет запускаться при каждом запуске системы;
  + подробная документация. Доступна официальная документация по данному драйверу;
  + логирование. В драйвере предусмотрена запись ошибок и промежуточных результатов в лог-файл.
* недостатки:
  + работа на уровне ядра. В драйвере ведется работа с файлами /proc, которые ядро создает для уровня пользователя, поэтому такая работа является не совсем корректной;
  + регистрация обработчиков событий NETDEV\_UP/NETDEV\_DOWN/NETDEV\_CHANGE от сетевых интерфейсов в неподходящий момент. Например, в системе может произойти событие NETDEV\_UP, а драйвер обработает его как событие NETDEV\_CAHNGE, что спровоцирует некорректную реакцию, так как выполнится неверный участок кода. Такое поведение может приводить к формированию неправильно маски;
  + невозможно запускать в интерактивном режиме (когда вся информация о работе модуля выводится на экран оператора). У данного драйвера существует только рабочий режим, отсутствуют какие-либо другие варианты запуска модуля.
* вывод по решению: ввиду того, описанных выше недостатков и того, что вся необходимая информация о прерываниях хранится в файлах пользовательского уровня, принято решение создать модуль для пользовательского уровня.

#### Драйвер igb

У компании Intel существует драйвер igb, который поддерживает все гигабитные сетевые соединения. Он предназначен для мониторинга сетевого трафика.

* достоинства:
  + есть официальная документация в открытом доступе, что позволяет оператору самостоятельно установить данный драйвер;
  + драйвер поддерживает концепцию NAPI, что позволяет работать с высокоскоростным трафиком;
  + логирование, но только в режиме отладки.
* недостатки:
  + драйвер осуществляет только мониторинг сетевых интерфейсов. Драйвер не осуществляет корректировки масок прерываний;
  + работа на уровне ядра. В драйвере ведется работа с файлами /proc, которые ядро создает для уровня пользователя, поэтому такая работа является не совсем корректной;
  + логирование происходит только в режиме отладки. На протяжении работы драйвер не создает лог-файлов, и оператор не может просмотреть промежуточные результаты работы.
* вывод по решению: драйвер не выполняет одну из ключевых функций заявленной в требованиях, а именно корректировку масок прерываний, поэтому данный драйвер не может использоваться на предприятии.

#### Демон irqbalance

Для ОС Linux разработано решение, которое автоматически распределяет нагрузку обработки прерываний по всем доступным CPU. Это решение – irqbalance.

Irqbalance определяет источники прерываний наибольшего объема и изолирует каждый из них на один CPU, чтобы нагрузка распределялась как можно равномерней по всем ядрам, при этом минимизируя пропуски кэша для обработчиков IRQ.

Установка данного демона производится с помощью команды «sudo apt install irqbalance».

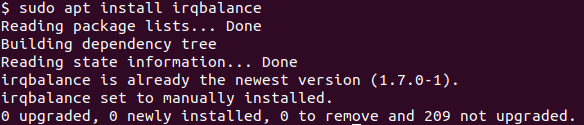


Рис. 1.7. Процесс установки irqbalance

Запустить данный демон можно с помощью команды «irqbalance -o», в данном случае демон совершает лишь одну итерацию и после этого завершает свою работу.

Помимо флага -o (--oneshot) для данного демона предусмотрен еще ряд опций:

* --debug – режим отладки, при котором будет выводиться дополнительная информация в терминал пользователю;

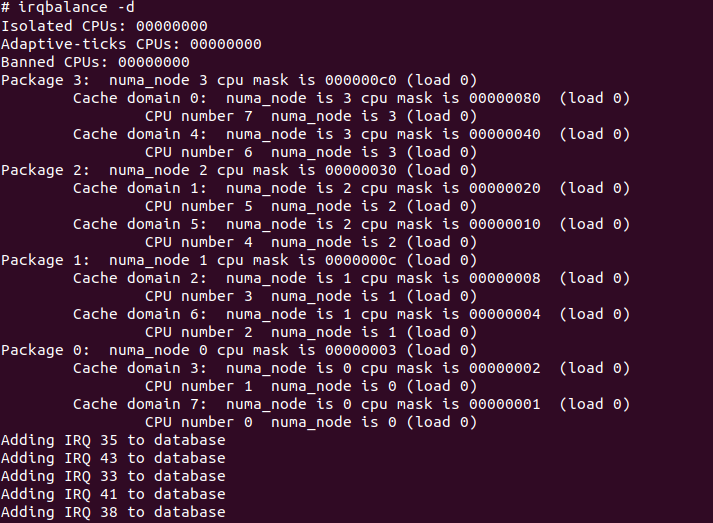


Рис. 1.8. Фрагмент информационных сообщений при запуске irqbalance с флагом -d

* --foreground – запускает irqbalance в интерактивном режиме;
* --banirq – с помощью данной опции задаются номера IRQ, которые демон будет игнорировать, изменения по этим прерывания производиться не будут;
* --pid – с помощью данной опции можно создать файл с номером PID процесса, под которым будет работать irqbalance.

Проведен анализ достоинств и недостатков данного демона:

* достоинства:
  + открытый исходный код. Данный проект является открытым, весь исходный код представлен на сайте GitHub. Каждый разработчик может просмотреть и изучить данный код;
  + автоматизированность. Данный демон самостоятельно распределяет нагрузку на ядра, от пользователя требуется только запустить программу;
  + irqbalance реализован в виде демона. Программа будет работать в фоновом режиме, позволяя пользователю работать параллельно с другими задачами;
  + логирование. Оператор может отследить, как выполнялась работа программы в лог-файлах;
  + наличие интерактивного режима. Позволяет оператору просмотреть, как работает программа;
  + работа на уровне пользователя. Является преимуществом, так как файлы /proc ядро создает именно для уровня пользователя.
* недостатки:
  + нет возможности работы с тестовым файлом с прерываниями. Работоспособность модуля придется сразу проверять на реальных данных, что может привести к сбоям в системе, если модуль отработает некорректно;
  + нет «сухого» режима. Данный режим также позволяет протестировать работу модуля перед внедрением его в систему;
  + нет возможности конфигурировать интерфейсы по именам. Irqbalance конфигурирует все интерфейсы в системе, IRQ номера которых не указаны в списке игнорируемых.
* вывод по решению: данное решение способно выполнять одну из ключевых задач для предприятия, а именно корректировка прерываний, но это решение в основном используется для быстрой базовой настройки прерываний сетевых интерфейсов и не подходит для более детальной настройки. Поэтому данное решение не может быть использовано в компании. Также нет возможности интегрировать имеющеюся документацию по организации выделения irq\_cores и working\_cores в продукт.

Все рассмотренные решения не удовлетворяют всем требованиям компании, поэтому принято решение о создании программного модуля мониторинга и корректировки прерываний сетевых интерфейсов, работающего на уровне пользователя.

## Концептуальная модель

В качестве типа программного модуля для уровня пользователя выбран тип демон. Демон – это сервисный процесс, который работает в фоновом режиме и контролирует систему. Такой тип выбран потому, что необходим модуль, который работал бы в фоновом режиме и с определенной периодичностью вызывал подпрограмму, которая конфигурирует прерывания для заданных интерфейсов.

#### Функциональная схема программного модуля

Функциональная схема призвана показать взаимодействие и состав компонентов в разрабатываемом решении. Данная схема отражает функционал ПМ МСИ.

Модуль может запускаться двумя способами:

1. При старте ОС (основной режим работы модуля). В таком режиме работают все подобные модули на предприятии. Именно поэтому для реализации выбран тип программы демон, так как демона можно легко «вшить» в ОС и настроить его запуск при старте системы.
2. Запуск оператором вручную. Это одно из требований в компании, так как необходим интерактивный режим работы, при котором можно отследить правильность и корректность работы модуля. Также наличие такого режима работы позволяет запустить этот модуль в любое время, вне зависимости от системы, где он используется.

Также данные о номерах IRQ могут браться из /proc/interrupts (основной источник) и из специально заготовленного файла (используется для режима отладки с помощью флага –t).

Эта функциональная схема демонстрирует, что разрабатываемый модуль является составным:

1. Управляющий сервис, который принимает аргументы командной строки, с помощью которых задается режим работы программы и задается путь до конфигурационных файлов. Также сервис осуществляет парсинг конфигурационного файла для определения интервала работы. Инициализируется лог-файл и файл с номером PID процесса, если модуль запускаться в режиме демона.
2. Модуль корректировки, так называемая, рабочая часть программы. Осуществляет формирование и установку масок прерываний для заданных сетевых интерфейсов.



Рис. 1.9. Функциональная схема ПМ МСИ

Также данная схема отражает функционал и взаимодействие сервиса и подпрограммы.

По данной схеме сформированы конкретные требования к ПМ МСИ:

* считывание входных данных:
  + считывание интервала работы модуля из файла;
  + считывание данных о сетевых интерфейсах из файла:
    - считывание названия;
    - считывание количества обрабатывающих ядер;
  + считывание входных аргументов из командной строки:
    - задание режимов работы:
      * интерактивный режим работы;
      * «сухой» режим работы;
      * режим работы, когда считывание данных о прерываниях задается из файла, а не из операционной системы;
      * путь до файла с прерываниями в случае, если задан флаг –t.
    - задания пути к конфигурационным файлам;
* формирование масок ядер процессора в каждой новой итерации работы модуля;
* формирование массива нума-нод процессора;
* формирование маски прерывания для сетевого интерфейса;
* установление маски прерывания для сетевого интерфейса путем записи в файл smp\_affinity;
* наличие управляющего сервиса;
* наличие рабочей части программы, которая будет формировать и записывать маску в файл smp\_affinity;
* работа в циклическом режиме с определенным интервалом работы, задаваемым в конфигурационном файле;
* возможность работы в фоновом режиме;
* возможность работы в интерактивном режиме для осуществления отладки и тестирования;
* запись всех промежуточных и итоговых результатов в лог-файлы.

## Схема данных программного модуля

Разрабатываемый ПМ будет запускаться при старте ОС, как и другие подобные модули на предприятии, без участия человека, но также этот модуль может быть запущен оператором вручную, например, для режима отладки работы модуля или проверки правильности его работы. Также такой режим работы позволяет достаточно легко внедрять новый функционал в разрабатываемый модуль, так как процедура тестирования новых функций за счет ручного режима упрощена.

На рисунке представлена схема данных, разработанная для ПМ МСИ по требованиям компании и описанной функциональной схеме, которая отражает работу решения, в случае его запуска вручную оператором. Отличия от случая работы модуля, когда он запускается при старте ОС, незначительные, пропускается лишь первый пункт (ввода с клавиатуры не осуществляется, так как нет прямого взаимодействия с пользователем). Модуль будет сразу инициироваться как программа-демон. Дальнейшие действия совпадают.



Рис. 1.10. Схема данных ПМ МСИ

Структура работы ПМ МСИ представляет собой следующую последовательность действий и событий:

1. Оператор запускает сервис с параметрами. Параметрами являются флаги работы и путь до конфигурационного файла. Флаги сервиса:
   * -f – флаг интерактивного режима работы. Все сообщения будут выводиться не только в специализированный лог-файл, но и на экран пользователя (не является обязательным параметром), программа не переходит в режим демона;
   * -t – флаг считывания данных о прерывании из заготовленного файла. Данный флаг позволяет отлаживать программу на тестовых данных, чтобы была возможность протестировать все варианты работы модуля перед его внедрением в реальную систему (не является обязательным параметром). При указании данного флага обязательно нужно указать путь до файла с прерываниями;
   * -d – флаг «сухого» режима работы. При данном режиме работы все сообщения поступают на экран пользователя, а также модуль не вносит никаких изменений в работу системы (не является обязательным параметром). При использовании данного флага также идет обязательным использование флага -f.
2. С заданной периодичностью, которая указана в конфигурационном файле, сервис запускает подпрограмму, которая конфигурирует маски прерывания.

Структура входных данных

В качестве входных данных у ПМ МСИ выступают:

* параметры работы модуля (флаги и путь к конфигурационным файлам);
* конфигурационный файл, в котором содержатся:
  + названия сетевых интерфейсов, маски прерывания, которых необходимо корректировать модулю (указываются на отдельных строках);
  + интервал работы сервиса в секундах.



Рис. 1.11. Пример конфигурационного файла сервиса

* файл с информацией о количестве ядер, выделяемых для обработки прерываний каждого интерфейса;

Структура данного файла представляет из себя набор строк следующего формата [\*]:<irq\_cores>/<working\_cores>:

* Параметр до «:» определяет выполнять или не выполнять привязку прерываний от сетевых интерфейсов к процессорам. Если указать «\*», тогда привязываются все прерывания интерфейсов. Сервис выставляет заданные в файле настройки в /proc/irq/<irq\_num>/smp\_affinity для этих интерфейсов. Если ничего не указать (то есть ничего перед символом «:»), тогда привязка прерываний к процессорам не выполняется, и прерывания сетевых интерфейсов распределятся в соответствии с алгоритмом назначения прерываний Linux. В таком случае файлы /proc/irq/<irq\_num>/smp\_affinity не будут модифицированы, что вызовет снижение производительности подсистемы обработки пакетов драйвером;
* <irq\_cores> – число процессорных ядер, полностью выделенных под обработку прерываний сетевых интерфейсов;
* <working\_cores> – количество рабочих ниток, число процессорных ядер, используемых для IPsec обработки.

IPsec – это группа интернет протоколов, которые используются для установки зашифрованных соединений между устройствами.



Рис. 1.12. Пример задания распределения irq\_cores, working\_cores для сетевых интерфейсов

Также в качестве входных данных может использоваться файл, в котором хранится информация о различных прерываниях, для тестирования работы программы. Данный файл используется при указании флага –t. Структура данного файла следующая:

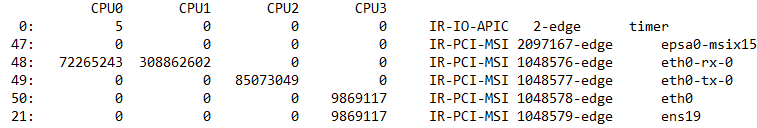


Рис. 1.13. Структура файла с прерываниями

Данный файл моделирует реальные данные, которые находятся в файле /proc/interrupts. А именно в нем содержится:

* самый левый столбец – номера irq прерываний интерфейсов;
* первая строка – ядра, которые находятся в системе;
* столбец под определенным ядром – количество прерываний, обработанных этим ядром от различных интерфейсов;
* правый столбец – имена сетевых интерфейсов.

Также в качестве входных данных модуля выступают данные взятые из потоков lscpu, /proc/cpuinfo, /proc/interrupts.

Структура выходных данных

Выходными данными является специальный лог-файл, в котором сохраняется промежуточная и результирующая информация и измененный файл smp\_affinity, в который модуль записал маску.

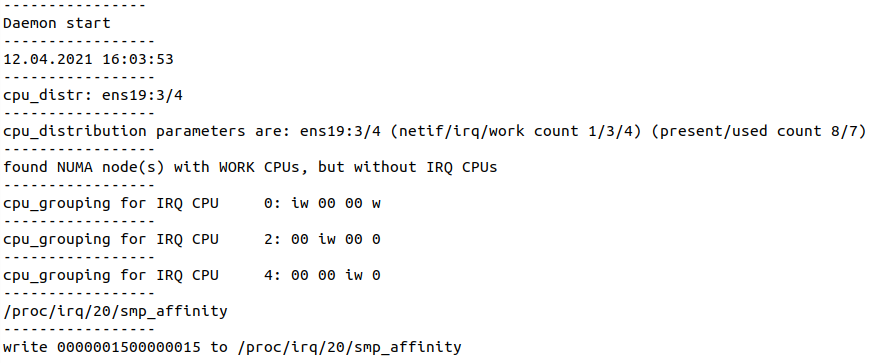


Рис. 1.14. Пример лог-файла

В данном лог-файле отражается следующая информация:

* дата и время начала работы сервиса;
* параметры корректировки (название интерфейсов и распределение irq\_cores / working\_cores);
* описание распределения нума нод;
* путь, куда будет записана маска прерывания;
* сообщение о том, что маска успешно записана в файл.

## Выводы по исследовательскому разделу

В разделе исследована предметная область и обоснована актуальность разработки ПМ МСИ. Выявлена проблематика данной области, сформированы цели и задачи разработки модуля. Рассмотрены аналогичные решения, выявлены их достоинства и недостатки.

Также разработана и проанализирована схема данных, структура входных и выходных данных, используемых в решении.

Представлена функциональная схема и сформированы требования к ПМ МСИ.

# КОНСТРУКТОРСКИЙ РАЗДЕЛ

## Схема алгоритма

Перед реализацией ПМ МСИ были рассмотрены другие решения, реализованные в компании на языке C. У всех этих решений имеется общая структура, а именно наличие разделения на управляющий сервис и рабочую часть программы. Управляющий сервис осуществляет контроль работы всей программы, а рабочая часть непосредственно выполняет тот функционал, который описан в требованиях.

#### Схема алгоритма сервиса

На рисунке приведена схема алгоритма управляющего сервиса.



Рис. 2.1. Схема алгоритма ПМ МСИ

Так как в исследовательском разделе описано, что типом программной реализации является демон, то необходимо рассмотреть особенности таких программ. Программа демон работает в фоновом режиме, отвязавшись от породившего ее терминала. При инициализации демона поток – родитель закрывает стандартные потоки (stdin, stdout, stderr), после чего выполняет команду fork(), которая создает новый поток, именуемый потомком. Номер PID созданного потомка записывается в специальный файл, для того чтобы оператор мог в любой момент завершить работу модуля по средствам вызова команды «kill». После поток-родитель закрывается, тем самым обрывая связь между потомком и терминалом. Таким образом, программа переходит в режим демона (фоновый режим). Также стоит отметить, что такая программа будет работать на уровне пользователя.

На схеме приведен алгоритм работы модуля, если его запускает оператор вручную, так как у модуля предусмотрен режим работы как ручной, так и запуск при старте ОС.

При запуске сервиса оператор указывает необходимые флаги (-f – интерактивный режим, -d – «сухой» режим, -t – работа с файлом с прерываниями, в случае указания данного флага, оператор обязан указать путь до данного файла, также оператор может и не указывать флаги, так как они не являются обязательными параметрами) и путь к конфигурационным файлам (путь к данным файлам является обязательным параметром, в случае, если оператор не укажет его, модуль выдаст ошибку в терминал).

Далее модуль осуществляет проверку наличия необходимых ему файлов, находящихся в директории, которую указал пользователь при старте модуля, если таких файлов не обнаружится, сервис выведет сообщение об ошибке и завершит работу.

Если конфигурационные файлы успешно прочитаны, сервис закрывает стандартные потоки (stdin, stdout, stderr), если не указан флаг –f при старте программы, в противном случае модуль оставляет их открытыми, для того чтобы выводить промежуточную и результирующую информацию в терминал.

Для того чтобы сервис мог прекратить свою работу в интерактивном режиме при помощи отправки сигнала SIGINT (нажатие CTRL+C) и с помощью команды kill, необходимо заблокировать данный сигнал, так как ожидание прихода сигнала осуществляется с помощью команды «sigtimedwait». Если сигнал не будет заблокирован, то будет вызван стандартный обработчик события для SIGINT, который сразу завершает программу, но перед завершением программы необходимо удалить все временные файлы и переменные, которые были созданы в ходе работы модуля.

После этого инициализируется бесконечный цикл, который с определенным периодом будет вызывать подпрограмму для выставления масок прерываний. Выход из данного сервиса возможен с помощью отправки процессу сигналов SIGINT, SIGKILL. Если сервис запущен не в интерактивном режиме, то оператор может узнать PID процесса из специального файла, который создается при инициализации демона, и затем может завершить процесс при помощи команды «kill -2 <PID>».

Структура решения, которое разделено на две части, позволяет сделать общий код, более читаемым, также позволяет разделить управление и работу программы.

По приходу сигнала о завершении работы цикл прерывается, освобождается вся динамически выделенная память, удаляется временный файл, который являлся индикатором работы сервиса, программа завершает свою работу.

#### Схема алгоритма подпрограммы

Приведена схема алгоритма подпрограммы, которая корректирует маски. Данная подпрограмма является отдельной самостоятельной программой, которая может использоваться для однократной корректировки масок.

Данная программа получает на вход такие же аргументы, как и сервис (флаги и путь к конфигурационным файлам). Выполняет проверку этих входных аргументов на соответствие шаблонам. Если аргументы не соответствуют, то подпрограмма выдаст ошибку и завершит работу, передав управление сервису, если программа была запущена с помощью сервиса.

После того как были считаны и проверены аргументы из командной строки, начинается работа подпрограммы. В самом начале формируется маска доступных (онлайн) CPU. Затем происходит формирование массива нума нод.

Далее начинается непосредственная обработка информации о распределении irq\_cores и working\_cores для сетевых интерфейсов. Происходит парсинг строки вида [\*]:<irq\_cores>/<working\_cores>. Определяется имя интерфейса, записываются в переменные значения irq\_cores и working\_cores.



Рис. 2.2. Схема алгоритма подпрограммы, выполняющей корректировку маски прерываний

После успешного парсинга всех таких строк, читается конфигурационный файл, в котором содержаться имена интерфейсов, за которыми необходимо следить сервису. Для каждого считанного интерфейса производится проверка, указано ли для него распределение irq\_cores и working\_cores, если такая строка в файле есть, то осуществляется парсинг данных, которые берутся либо из файла с прерываниями (если при старте программы указан флаг –t и путь к данному файлу), либо из файла /proc/interrupts. По имени интерфейса определяется соответствующее ему номер IRQ прерывания, которое будет использовано для записи в нужный файл smp\_affinity.

По завершении успешного парсинга /proc/interrupts или специального файла с прерываниями, осуществляется запись в файл smp\_affinity, расположенный по пути /proc/irq/<прочитанный номер прерывания>/smp\_affinity.

Результатом работы программы является измененная маска прерывания, записанная в файл smp\_affinity.

## Анализ языков программирования

Перед реализацией ПМ МСИ был проведен анализ языков программирования, для того чтобы определить наиболее подходящий язык для разработки. Сформированы требования для языка, исходя из функциональных требований к модулю. Для рассмотрения выбраны несколько популярных языков программирования. Результаты данного анализа представлены в таблице 2.1.

Табл. 2.1.

Анализ языков программирования

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Языки  программирования  Критерии | C [1] | Java [3] | C# [2] | C++ [1] |
| Используемый язык в компании С-Терра | Является одним из основных языков в компании | Является одним из основных языков в компании | Не является одним из основных языков в компании | Является одним из основных языков в компании |
| Тип исполнения | Компилируемый | Компилируемый - интерпретируемый | Компилируемый - интерпретируемый | Компилируемый |
| Возможность получения информации о нума нодах | Есть | Есть | Есть | Есть |
| Возможность получения информации о ядрах процессора | Есть | Есть | Есть | Есть |
| Возможность создания программы, работающей в фоновом режиме | Есть | Есть | Есть | Есть |

Продолжение Табл. 2.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Возможность вызова сторонней подпрограммы из кода | Есть | Есть | Есть | Есть |

Источники информации:

[1] C/C+. Программирование на языке высокого уровня [Текст] : Учебник для вузов / Павловская Т.А.. - СПб. : Питер, 2005, 2001  
[2] http://msdn.microsoft.com/ru-ru/vcsharp/default.aspx

[3] http://www.oracle.com/technetwork/java/index.html

Язык Си разработан американцем Деннисом Ритчи в исследовательском центре Computer Science Research Center of Bell Laboratories корпорации AT&T в 1972 г. Первоначальная реализация Си была выполнена на ЭВМ PDP-11 фирмы DEC для создания операционной системы UNIX. Си – это универсальный язык, который позволяет разрабатывать как приложения пользовательского уровня, так и приложения, предназначенные для работы на уровне ядра.

Язык C++ является потомком языка С, он, как и Си является типизированным языком, который поддерживает ООП, в отличие от Си, так и процедурное программирование. Последним стандартом является стандарт C++20, выпущенный в декабре 2020 года.

Язык C# современный объектно-ориентированный и типобезопасный язык программирования. C# позволяет разработчикам создавать множество типов безопасных и надежных приложений, работающих в экосистеме .NET. C# относится к широко известному семейству языков C, и покажется хорошо знакомым любому, кто работал с C, C++, Java или JavaScript. C# — это объектно- и компонентно-ориентированный язык программирования.

Язык Java на данный момент является одним из самых популярных языков программирования. Первая версия языка появилась еще в 1996 году в компании Sun Microsystems, впоследствии поглощенной компанией Oracle. Java задумывался как универсальный язык программирования, который можно применять для различного рода задач. На данный момент актуальным является версия Java 16, выпущенная в марте 2021 года. В основном данный язык используется для создания приложений связанных с графическими интерфейсами, такими как декстопные приложения для компьютеров, мобильные приложения под систему Android.

На выбор языка сильно повлияло то, что в компании для создания подобных сервисов используют язык С.

Также одним из главных условий являлась возможность создания программы-демона, так как эта программа работает на уровне пользователя, что является лучшим вариантом при работе с файлами /proc.

Также программа разрабатывается под операционную систему Linux, а язык Си лучше всего подходит для такой разработки, так как все ядро ОС написано на языке Си.

По результатам анализа выбран язык C, так как он удовлетворяет всем требованиям, предъявляемых для разработки ПМ МСИ.

## Анализ сред разработки

После того как выбран язык программирования, проведено исследование сред программирования, в которых возможно создание решений на языке C.

Табл. 2.2.

Анализ сред программирования

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Среды  разработки  Критерии | Visual Studio Code [2] | Visual Studio 19 [1] | Sublime Text 3 [3] | Eclipse [4] | Atom [5] |
| Возможность работы в ОС Linux | ОС Linux поддерживается | ОС Linux не поддерживается | ОС Linux поддерживается | ОС Linux поддерживается | ОС Linux поддерживается |
| Встроенный отладчик | Есть | Есть | Есть | Есть | Есть |
| Форма распространения ПО | Бесплатно | Бесплатно (неполный функционал) | Бесплатно | Бесплатно | Бесплатно |

Продолжение Табл. 2.2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Поддержка контроля версий Git | Поддерживает, встроено в среду | Поддерживает, встроено в среду | Поддерживает, встроено в среду | Поддерживает, встроено в среду | Поддерживает, встроено в среду |
| Встроенный терминал | Есть | Есть | Нет | Есть | Есть |
| Возможность создания приложения на языке С | Есть | Есть | Есть | Есть | Есть |

Источники информации:

[1] https://visualstudio.microsoft.com/ru/vs/

[2] https://code.visualstudio.com/

[3] https://www.sublimetext.com/docs/3/linux\_repositories.html

[4] https://www.eclipse.org/

[5] https://atom.io/

Visual Studio Code - это легкий, но мощный редактор исходного кода, который работает на вашем рабочем столе и доступен для Windows, macOS и Linux. Он поставляется со встроенной поддержкой JavaScript, TypeScript и Node.js и имеет богатую экосистему расширений для других языков (таких как C ++, C #, Java, Python, PHP, Go) и среды выполнения (например, .NET и Unity).

Visual Studio 2019 среда разработки, которая предоставляет полный набор инструментов для всех этапов разработки, от начального замысла до финального развертывания. Позволяет создавать множество различных типов приложения от консольных до веб-приложений. Поддерживает несколько языков программирования.

Sublime Text 3 является редактором кода, позволяющий писать код на многих языках программирования. Доступен для работы и на ОС Mac, и на ОС Windows и, самое главное, на ОС Linux.

Eclipse – свободная интегрированная среда разработки модульных кроссплатформенных приложений. Развивается и поддерживается Eclipse Foundation. В основном использует для разработки на языке Java, но также поддерживает разработку и на других языках, например, Си. Имеет бесплатный и открытый исходный код.

Atom – это декстопное приложение, созданное с интеграцией HTML, JavaScript, CSS и Node.js. Он работает на Electron, фреймворке для создания кроссплатформенных приложений с использованием веб-технологий. При этом позволяет разрабатывать не только веб-приложения, но и другие виды приложений.

Одним из главных критериев выбора среды программирование – это критерий возможности работы среды в ОС Linux, так как данный модуль предназначен для Linux.

Также важным условием являлось то, что присутствует поддержка работы контроля версий Git, так как разработка проекта производится в общей системы компании.

Немаловажным аспектом также являлось то, что в среде присутствует встроенный терминал, который упрощает работу.

По результатам исследования выбрана среда разработки Visual Studio Code, так как она удовлетворяет всем требованиям, выдвинутым к разработке модуля.

## Анализ кода ПМ МСИ

#### Анализ кода управляющего сервиса

Вся программа начинается с обработки входных параметров, которые считываются с командной строки.

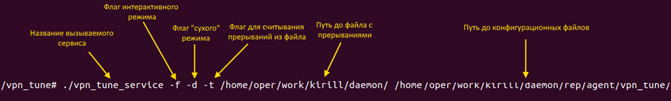


Рис. 2.3. Пример аргументов командной строки

if(argc < 2)

goto usage;

const char \*path\_to\_interrupts\_file = NULL;

int interactive\_mode = 0;

int dry\_run\_mode = 0;

const char \*path = NULL;

int i = 1;

for (; i < argc; i++) {

if(!strcmp(argv[i], "-f"))

interactive\_mode = 1;

else if(argv[i][0] == '-' && argv[i][1] == 't') {

if(path\_to\_interrupts\_file) {

report\_err("Invalid arguments: duplicate path to file with interrupts\n");

goto usage;

}

if(argv[i][2])

path\_to\_interrupts\_file = &argv[i][2];

else if(++i == argc) {

report\_err("Invalid arguments: option '-t' requires an argument\n");

goto usage;

} else

path\_to\_interrupts\_file = argv[i];

}

else if(!strcmp(argv[i], "-d"))

dry\_run\_mode = 1;

else if(!path)

path = argv[i];

else {

report\_err("Invalid arguments: duplicate path to config files\n");

goto usage;

}

}

if(!path) {

report\_err("Invalid arguments: path to config files was not specified\n");

goto usage;

}

if(dry\_run\_mode && !interactive\_mode) {

report\_err("Invalid arguments: dry-run mode can be used only in interactive mode\n");

goto usage;

}

return run\_main(path, interactive\_mode, path\_to\_interrupts\_file, dry\_run\_mode);

usage:

fprintf(stderr, "Usage: %s [-f [-d]] [-t <file>] path\n", argv[0]);

fprintf(stderr, " where\n");

fprintf(stderr, "-f - enable interactive mode\n");

fprintf(stderr, "-d - enable dry-run mode\n");

fprintf(stderr, "-t <file> - read interrupts from <file>\n");

fprintf(stderr, "path - path to config files\n");

В данном примере кода, показана обработка входных аргументов. Аргументы можно вводить в любом порядке, программа их распознает. В случае возникновения ошибки с помощью функции fprintf(); пользователю будет сообщено, какую ошибку он допустил. Также будет показан пример использования флагов.

Если проверка аргументов прошла успешно, эти аргументы передаются в основную функцию сервиса run\_main();

В начале данной функции происходит конфигурирование режима работы сервиса. При наличии флага интерактивного режима программа не будет переходить в фоновый режим, а продолжит работать в терминале. Если флаг –f не указан, то тогда происходит инициализация демона.

Но перед инициализацией демона, происходит создание лог-файла, в который помещается информация о том, когда сервис начал свою работу и вся промежуточная информация о работе модуля.

После инициализации лог-файла считывается данные из конфигурационного файла, а именно интервал работы сервиса.

В случае ошибки при считывании интервала работы сервиса в функцию будет возращено отрицательное значение, также ошибка будет сопровождена сообщением, которое будет выведено в терминал оператору и продублировано в лог-файл. После этого сервис завершит свою работу.

Инициализация демона начинается с закрытия стандартных потоков ввода/вывода.

int i;

for(i = 0; i < 100; i++) close(i);

if(open("/dev/null", O\_RDWR)< 0) {

perror("./vpn\_tune\_service");

return 1;

}

dup(0); dup(0);

После того как потоки были закрыты, процесс-родитель создает потомка, в котором будет выполняться весь остальной код программы. При создании потомка его PID номер записывается в специализированный файл, для того чтобы данный процесс всегда можно было завершить. Также происходит изменение рабочей директории.

pid = fork();

if(pid < 0) {

return -1;

}

if(pid > 0) {

write\_pid(pid);

exit(EXIT\_SUCCESS);

}

umask(0);

char result[MAIN\_FILE\_BUFFER\_SIZE];

memset(result, 0, MAIN\_FILE\_BUFFER\_SIZE);

ssize\_t count = readlink("/proc/self/exe", result, MAIN\_FILE\_BUFFER\_SIZE);

const char \*path\_current\_exe\_file;

if(count != -1) {

path\_current\_exe\_file = dirname(result);

}

if(chdir(path\_current\_exe\_file) < 0)

return -3;

sid = setsid();

if(sid < 0) {

return -2;

}

Затем происходит блокировка сигнала SIGINT, так как интервальная работа сервиса обеспечивается функцией sigtimedwait(). Блокировка выполняется для того, чтобы заблокировать использование стандартного обработчика, так как стандартный обработчик вызывает немедленное завершение программы, что не позволяет корректно завершить работу программы.

sigset\_t set, orig;

sigemptyset(&set);

sigaddset(&set, SIGINT);

sigemptyset(&orig);

sigprocmask(SIG\_BLOCK, &set, &orig);

Основная работа модуля кроется в следующем цикле.

while(1) {

#ifdef \_\_linux\_\_

while(1) {

system(work\_cmd);

sigtimedwait(&set, &info, &timeout);

if (SIGINT == info.si\_signo) {

break;

}

}

#else

#endif

free(work\_cmd)

return stop\_main();

}

В данном цикле с помощью команды system(); с заданной периодичностью вызывается подпрограмма, путь до которой хранится в переменной work\_cmd. Ожидание сигнала SIGINT происходит благодаря функции sigtimedwait();. При появлении сигнала, в поле si\_signo переменной info записывается номер сигнала, затем происходит проверка является ли пришедший сигнал SIGINT, если проверка пройдена, то цикл прерывается и высвобождается вся динамически выделенная память и программа завершает свою работу.

#### Анализ кода подпрограммы

Старт подпрограммы совпадет со стартом сервиса, происходит парсинг входных аргументов командной строки.

После успешного парсинга начинается основная работа подпрограммы. В функции work\_daemon(); инициализируется и создается маска ядер процессора, массив нума нод, производиться парсинг файла с распределением irq\_cores / working\_cores, запускается процесс формирования маски прерывания и ее записи в smp\_affinity.

Далее подробно будет проанализирован каждый из пунктов.

Первым действием является формирование маски ядер процессора.

count\_online\_cpus = 0;

count\_present\_cpus = 0;

cpu\_online\_mask = (struct cpumask \*)malloc(sizeof(struct cpumask));

cpu\_present\_mask = (struct cpumask \*)malloc(sizeof(struct cpumask));

for (int i = 0; i <

sizeof(cpu\_present\_mask->bits)/sizeof(cpu\_present\_mask->bits[0]);

i++) {

cpu\_present\_mask->bits[i] = 0;

}

for (int i = 0; i <

sizeof(cpu\_online\_mask->bits)/sizeof(cpu\_online\_mask->bits[0]); i++) {

cpu\_online\_mask->bits[i] = 0;

}

FILE \*const f = popen("grep processor /proc/cpuinfo", "r");

if(f == NULL){

return;

}

char s[256];

const int n = 256;

int k = 0;

int cpu = 0;

while(fgets(s, n, f) != NULL) {

if(k % 2 == 0) {

cpu = parse\_number\_from\_cpu\_info(s);

cpumask\_set\_cpu(cpu, cpu\_present\_mask);

cpumask\_set\_cpu(cpu, cpu\_online\_mask);

count\_online\_cpus++;

count\_present\_cpus++;

}

k++;

}

fclose(f);

Процессоры, которые используются в компьютерах компании, имеют несколько ядер. Получение информации об онлайн CPU происходит с помощью команды /proc/cpuinfo. Эта команда позволяет узнать количество онлайн CPU на момент вызова команды, что позволяет сформировать маску. Информацию о действительно доступных CPU (онлайн) необходимо получать именно в подпрограмме, так как сервис может работать неограниченное количество времени (пока оператор не завершит его работу вручную или пока систему не выключат), ядра могут отключаться в течение работы сервиса, поэтому необходимо учитывать данный факт при формировании маски.

Маска, которая будет сформирована, имеет вид 16-тиричного числа, которое в двоичном представлении будет иметь единицы на позициях, соответствующих номерам CPU в системе.

При парсинге информации /proc/cpuinfo, программа запоминает номера CPU, которых указаны в конце строки, чтобы потом установить бит в единицу в том же месте в маске.

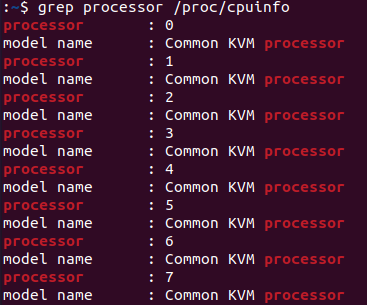


Рис. 2.4. Результат выполнения команды grep processor /proc/cpuinfo

Маска сохраняется в глобальной переменной cpu\_online\_mask.

Формат типа маски следующий:

struct cpumask {

cpu\_mask\_bits\_type\_t bits[(NR\_CPUS + sizeof(cpu\_mask\_bits\_type\_t)\*8 - 1)/

sizeof(cpu\_mask\_bits\_type\_t)\*8];};

Где NR\_CPUS – это максимальное количество CPUs, обрабатываемых программой.

После формирования маски онлайн CPU, формируется массив нума нод.

FILE \*const f = popen("lscpu", "r");

if(f == NULL) {

return;

}

char s[256];

int n = 256;

int k = 0;

int nodes = 0;

int last\_str = 0;

while(fgets(s, n, f) != NULL) {

if(check\_str\_to\_numa(s)) {

nodes++;

last\_str = k;

}

k++;

}

/\*

lscpu has an extra str, which starts with "NUMA node".

\*/

cpu\_node\_map = (int \*)malloc((sizeof(int)) \* (NR\_CPUS));

numa\_nodes\_count = nodes - 1;

for (int i = 0; i < NR\_CPUS; i++) {

cpu\_node\_map[i] = -1;

}

fclose(f);

FILE \*const f1 = popen("lscpu", "r");

if(f1 == NULL) {

return;

}

for (int i = 0; i < last\_str - nodes + 2; i++) {

fgets(s, n, f1);

}

for (int i = 0; i < nodes - 1; i++) {

fgets(s, n, f1);

set\_cpu\_to\_node(s, i);

}

fclose(f1);

Формирование массива нума нод осуществляется с помощью прочтения данных из потока lscpu. Прочтение данных производится с помощью команды popen(), которая создает процесс и вызывает оболочку, для выполнения команд. В качестве аргументов popen принимает команду, информацию из которой пользователь хочет получить и редим работы (r – режим чтения данных, w – режим записи данных).

В случае ошибке открытия потока функция вернет NULL (например, свободной памяти не хватает для выполнения функции).

Команда lscpu собирает информацию об архитектуре процессора из sysfs и /proc/cpuinfo. Sysfs – это виртуальная файловая система, которая используется в Linux. Вывод команды может быть оптимизирован для анализа или для удобства чтения людьми.

Эта информация включает, например, количество процессоров, потоков, ядер, сокетов и узлов с неоднородным доступом к памяти (NUMA).

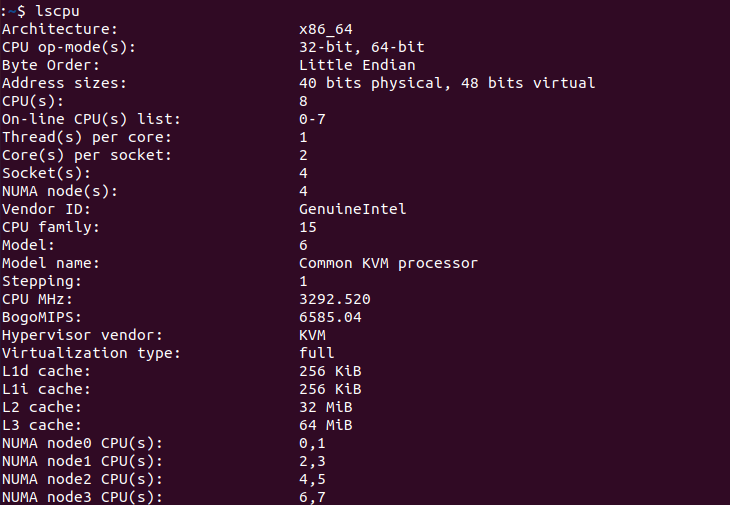


Рис. 2.5. Результат выполнения команды lscpu

После записи массива нума нод и маски CPU, программа выполняет парсинг файла cpu\_distr, в котором содержится информация о распределении irq\_cores / working\_cores для каждого интерфейса (в формате [\*]:<irq\_cores>/<working\_cores>).

ssize\_t n;

char \*cpu\_distr\_path = (char \*)malloc(strlen(path\_to\_main\_dir) + strlen(VPNDRVR\_CPU\_DISTR\_PARAM\_FILE) + 2);

if(cpu\_distr\_path == NULL) {

fprintf(stderr, "Memory allocation error\n");

return -2;

}

sprintf(cpu\_distr\_path, "%s/%s", path\_to\_main\_dir, VPNDRVR\_CPU\_DISTR\_PARAM\_FILE);

FILE \*const f = fopen(cpu\_distr\_path, "r");

char s[5000];

memset(s, 0, 5000);

if(f == NULL) {

sprintf(s, "Can't open %s for reading\n", VPNDRVR\_CPU\_DISTR\_PARAM\_FILE);

write\_log(s);

if(interactive\_mode) {

fprintf(stderr, "Can't open %s for reading\n", VPNDRVR\_CPU\_DISTR\_PARAM\_FILE);

}

return -1;

}

n = fread(cpu\_distr, sizeof(char), 4096, f);

fclose(f);

memset(s, 0, 5000);

if (n <= 0 || (size\_t)n >= sizeof(cpu\_distr)) {

sprintf(s, "Failed to read %s\n", VPNDRVR\_CPU\_DISTR\_PARAM\_FILE);

write\_log(s);

if(interactive\_mode) {

fprintf(stderr, "Failed to read %s\n", VPNDRVR\_CPU\_DISTR\_PARAM\_FILE);

}

cpu\_distr[0] = '\0';

return -1;

}

/\* trim trailing newlines \*/

while (n && cpu\_distr[n - 1] == '\n')

n--;

cpu\_distr[n] = '\0';

memset(s, 0, 5000);

sprintf(s, "cpu\_distr: %s", cpu\_distr);

write\_log(s);

if(interactive\_mode) {

printf("cpu\_distr: %s\n", cpu\_distr);

}

free(cpu\_distr\_path);

По завершении парсинга информации о прерываниях, irq\_cores и working\_cores программа переходит к считыванию информации об интерфейсах, маски которых необходимо модифицировать. Считывание происходит из конфигурационного файла, где в каждой новой строке, за исключением первой строки, указаны имена сетевых интерфейсов.

char \*config\_path = (char \*)malloc(strlen(path\_to\_main\_dir) + strlen("/config.txt") + 1);

if(config\_path == NULL) {

fprintf(stderr, "Memory allocation error\n");

return -2;

}

sprintf(config\_path, "%s/config.txt", path\_to\_main\_dir);

FILE \*const f = fopen(config\_path, "r");

if(f == NULL) {

printf("Error reading from file\n");

return -1;

}

char s[256];

int n = 256, k = 0;

fgets(s, n, f);

while(fgets(s, n, f) != NULL) {

k++;

}

fclose(f);

free(config\_path);

return k;

В качестве возвращаемого значения используется количество прочитанных интерфейсов, чтобы можно было провести проверку на наличие интерфейсов, и осуществлять корректировку масок только в том случае, если число интерфейсов в файле больше 0.

Приведен анализ кода, который модифицирует маску сетевого интерфейса.

char \*config\_path = (char \*)malloc(strlen(path\_to\_main\_dir) + strlen("/config.txt") + 1);

if(config\_path == NULL) {

fprintf(stderr, "Memory allocation error\n");

return NULL;

}

sprintf(config\_path, "%s/config.txt", path\_to\_main\_dir);

FILE \*const f = fopen(config\_path, "r");

if(f == NULL)

return NULL;

char s[256];

int n = 256, k = 0;

struct net\_device \*dev = (struct net\_device \*)malloc((sizeof(struct net\_device)) \* count);

fgets(s, n, f);

while(fgets(s, n, f) != NULL) {

dev[k].name = create\_interface\_name(s);

pcap\_linux\_topology\_netdev\_bind\_irqs(&dev[k]);

k++;

}

fclose(f);

free(config\_path);

return dev;

В данной функции происходит формирование переменной dev типа struct net\_device.

struct net\_device {

char \*name;

};

Ниже представлен фрагмент функции, которая определяет номер IRQ прерывания для каждого заданного сетевого интерфейса.

struct pcap\_topology\_parts parts;

struct pcap\_topology\_line \*line;

int ret = -ENOMEM;

if(path\_to\_interrupts\_file) {

char \*const path\_cpu\_distr = (char \*)malloc(strlen(

path\_to\_interrupts\_file) + strlen("/cpu\_distr.txt") + 1);

if(!path\_cpu\_distr)

return -1;

sprintf(path\_cpu\_distr, "%s/cpu\_distr.txt", path\_to\_interrupts\_file);

line = pcap\_linux\_init\_line(path\_cpu\_distr,

PCAP\_TOPOLOGY\_LINESZ);

} else {

line = pcap\_linux\_init\_line(PCAP\_TOPOLOGY\_IRQPATH,

PCAP\_TOPOLOGY\_LINESZ);

}

if (!line) {

return -ENOMEM;

}

/\* Skip header line. \*/

if (pcap\_linux\_topology\_read\_line(line))

goto err;

while (!pcap\_linux\_topology\_read\_line(line)) {

if (pcap\_linux\_topology\_parse\_line(line, &parts))

continue;

pcap\_linux\_topology\_parse\_netdev(line, &parts, nd);

}

ret = 0;

err:

pcap\_linux\_destroy\_line(line);

return ret;

Функция находит нужную строку по имени интерфейса и читает номер IRQ, который указан в начале строки.

На вход функция получается строку следующего вида:

48: 0 163 0 0 0 0 0 0 IR-PCI-MSI 1048576-edge ide2, eth0-rx-0, mga@pci:0000:01:00.0, где 48 – это номер IRQ прерывания, числа после двоеточия – это количество пакетов обработанных каждым ядром от этого прерывания, после идет название интерфейса и очереди RX/TX.

RX / TX очереди – это очереди, которые отвечают за прием и передачу пакетов у сетевых интерфейсов.

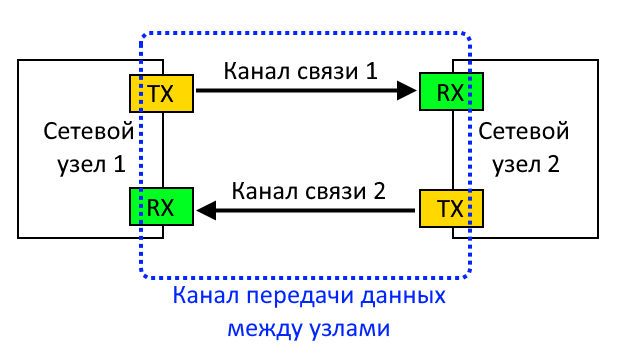


Рис. 2.6. Пример организации передачи данных между узлами

Для того чтобы определить номер IRQ программа парсит либо специализированный файл с прерываниями, в случае указания соответствующего флага (-t), либо информацию из /proc/interrupts.

Если парсинг прошел успешно (переменной ret присвоено значение 0), то будет осуществлено преобразование маски для записи и запись ее в smp\_affinity.

Запись в файл smp\_affinity осуществляется по пути /pcap/irq/<irq\_num>/smp\_affinity, если не задан «сухой» режим работы, так как в таком режиме программа не вносит никаких изменений в работу системы. Если оператор указал «сухой» режим, то сформированная маска отразится на консоли оператора, как и сформированная консольная команда, с помощью которой оператор может скорректировать маску вручную.

По завершении записи маски в smp\_affinity подпрограмма завершает свою работу и передает управление сервису, который вызовет эту подпрограмму через заданный интервал времени.

В качестве интерфейса программного модуля выступает интерфейс консольного приложения, так как основной режим работы ПМ – это фоновый режим демона. Для использования модуля вручную запуск осуществляется путем ввода названия исполняемого файла и указания необходимых флагов и пути до конфигурационных файлов. Если пользователь укажет интерактивный режим работы, то ему будет выводиться вся информация в терминал.

Приведен пример запуска ПМ МСИ оператором вручную. В данном примере использованы все флаги, а именно флаг интерактивного режима (-f), флаг «сухого» режима (-d), флаг –t, означающий считывание информации о прерываниях из специально подготовленного файла, далее идет путь до файла с прерываниями и путь до конфигурационных файлов. Такой набор флагов позволяет проверить корректность формирования маски прерывания и пути, куда будет записана маска. После проверки корректности оператор может начать использовать модуль в основном режиме работы (без указания флагов).

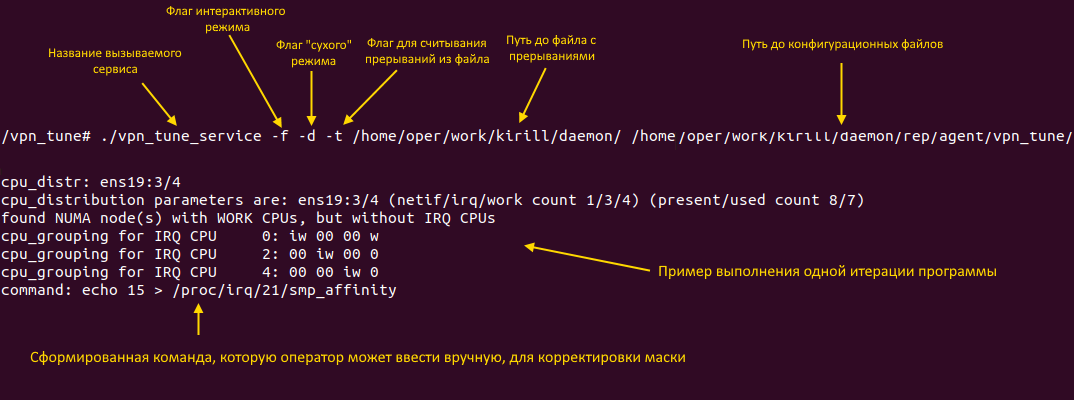


Рис. 2.7. Пример запуска ПМ МСИ с использованием флагов

## Выводы по конструкторскому разделу

В разделе проведен анализ языков и сред программирования, выбран язык, удовлетворяющий всем требованиям, и для этого языка подобрана среда разработки.

Также описаны ключевые фрагменты кода ПМ МСИ и приведен пример использования модуля в системе.

# ТЕХНОЛОГИЧЕСКИЙ РАЗДЕЛ

## Сборка программ на языке Си

Сборка программы написанной в рамках данной работы осуществлялось с помощью компилятора gcc. Компилятор gcc предназначен для сборки программ, написанных на языке Си. Он является бесплатным и кроссплатформенным.

gcc может компилировать программы, написанные на различных ЯП:

* Си;
* C++;
* Objective-C;
* Fortran;
* Ada;
* Go.

Данный компилятор активно развивается и поддерживается в настоящее время.

При сборке с помощью gcc можно указывать различные опции, которые будут изменять параметры компиляции:

* -c – данная опция является основной, так как с ее помощью отмечаются исходные файлы, в которых содержится код;
* -o – с помощью данной опции указываются выходные исполняемые файлы;
* -g – данная опция отвечает за включение отладочной информации для отладчика GDB;
* -Wall – опция, которая включает вывод информационных сообщений об ошибках в процессе компиляции;
* -pedantic – позволяет получить информацию об ошибках, связанных с нарушением использования стандартов.

При компиляции ПМ МСИ использовались все перечисленные опции:



Рис. 3.1. Компиляция ПМ МСИ

## Отладка ПМ МСИ

Важным этапом разработки любого программного продукта является отладка его работы. С помощью отладки программист в ручном режиме может проверить работу основных узлов проекта и их взаимодействие между собой. Также при наличии ошибок в работе модуля, отладка позволяет обнаружить место в коде, где работа модуля нарушается.

Произведен анализ средств отладки, которые использовались при реализации ПМ МСИ.

#### Отладчик GDB

Отладчик – это программа, которая запускает и исполняет программу, которую программист указывает при старте отладчика, внутри себя. Основной задачей отладчика является предоставление средств по контролю выполнения программы программисту, то есть предоставлять доступ к пошаговому исполнению программы, отслеживанию значений переменных в каждый момент времени. Наиболее популярным отладчиком для операционной системы Linux является программа GNU GDB. GDB имеет широкий функционал, который позволяет проводить отладку ПО в полном объеме, но для простой отладки достаточно использовать лишь некоторые из возможных функций.

Отладчик GDB может быть представлен в разных видах:

* в терминале Linux;
* в среде программирования.

Основными командами при работе с отладчиком GDB в терминале являются:

* run (r) – с помощью этой команды начинается отладка программы;
* next (n) – данная команда совершает переход к следующей строке кода;
* breakpoint (b) – с помощью этой команды устанавливается точка остановы на определенной строке (например, «b 418»);
* quit (q) – эта команда завершает отладку;
* print – с помощей данной команды можно узнать значение какой-либо переменной (например, «print argc»);
* continue (c) – переход к следующей точке останова;
* finish – завершает выполнение функции.

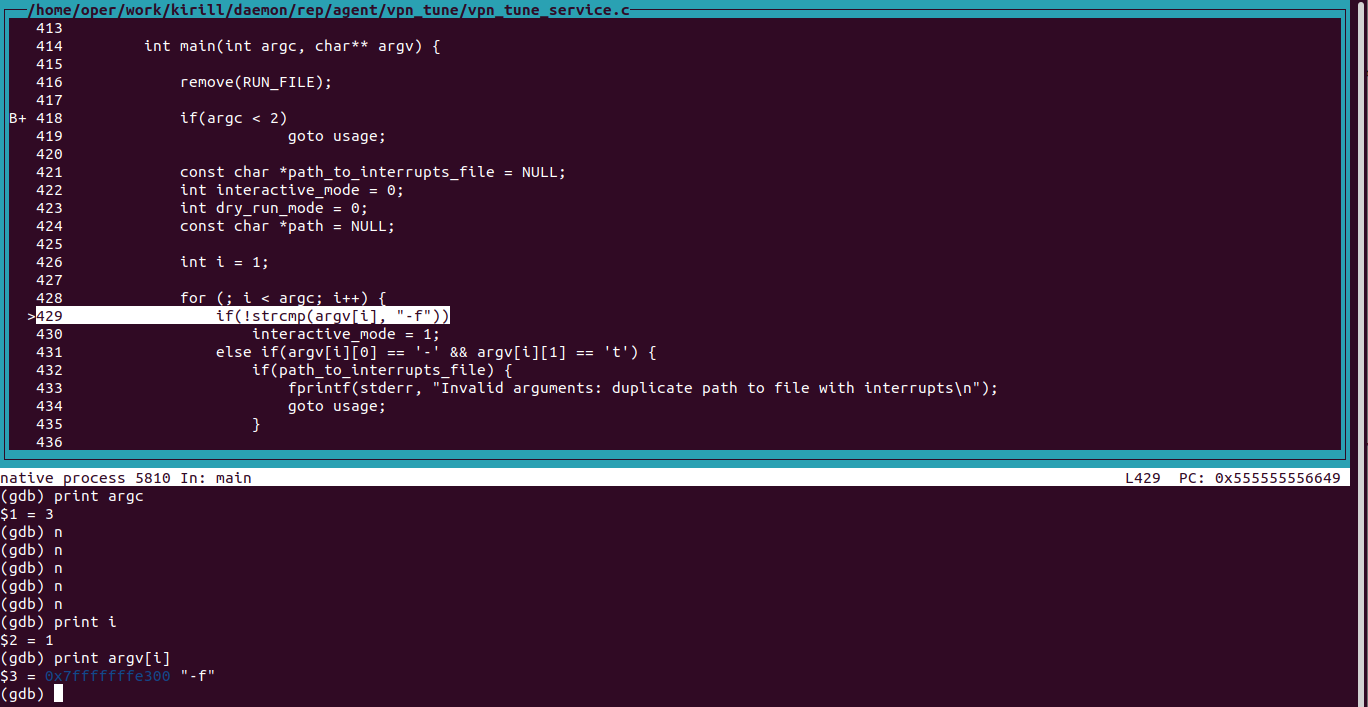


Рис. 3.2. Экранная форма отладки с помощью GDB в терминале

Языки программирования, которые поддерживает GDB:

* Ada;
* Assembly;
* C;
* C++;
* D;
* Fortran;
* Go;
* Objective-C;
* OpenCL;
* Modula-2;
* Pascal;
* Rust.

В данной работе выбран способ отладки с помощью среды разработки Visual Studio Code. В ней используется тот же отладчик GDB только в более удобной и наглядной оболочке.

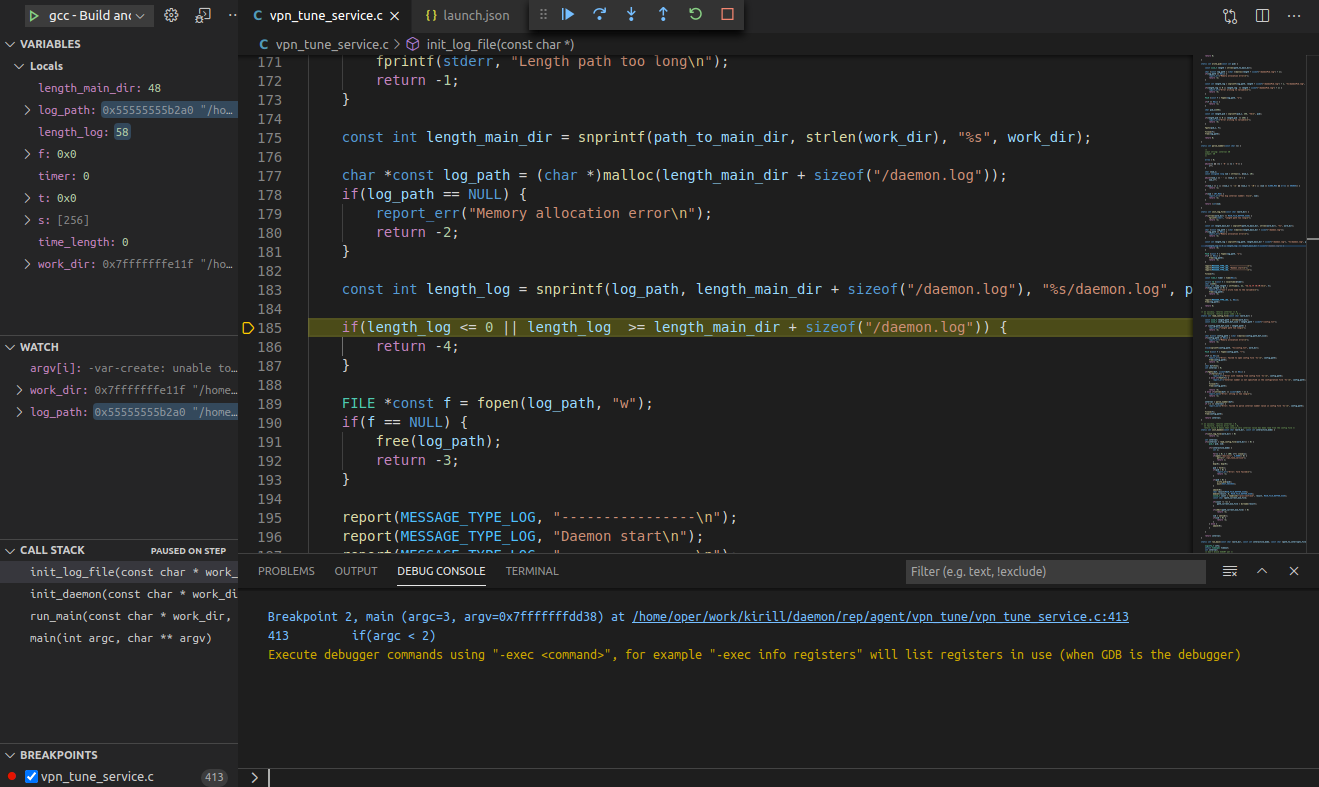


Рис. 3.3. Окно отладки в среде VS Code

На рисунке приведена экранная форма отладки в среде Visual Studio Code с помощью отладчика GDB.

Проведен анализ всех областей отладки.

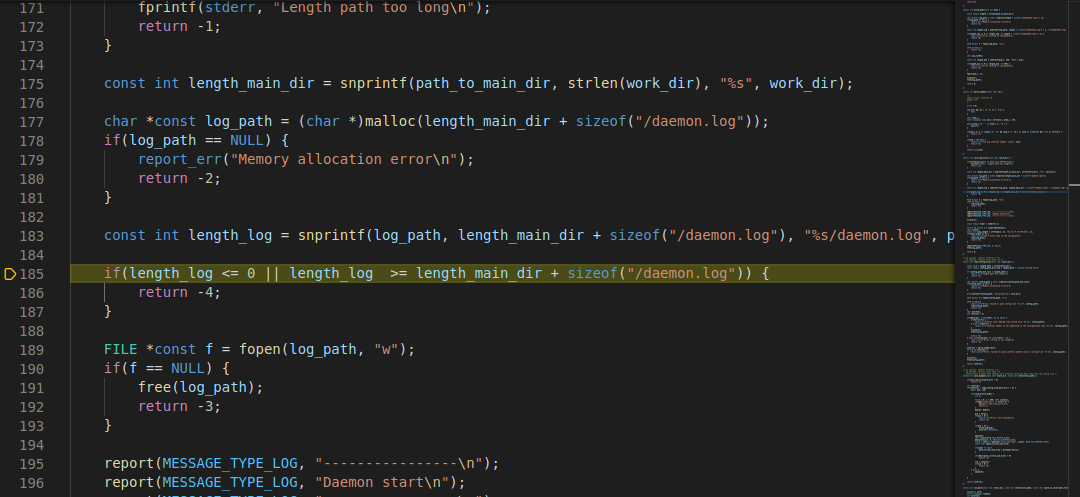


Рис. 3.4. Основное окно отладки с исходным кодом

Представленное окно содержит исходный код программы с обозначением точек останова, обозначение текущей строки выполнения программы.

Переход к следующим строкам кода происходит с помощью команд:

* step over (F10);
* step into (F11);
* step out (Shift+F11).

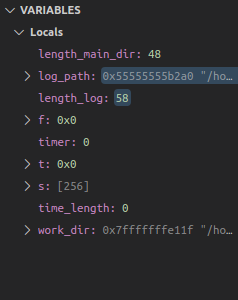


Рис. 3.5. Окно с локальными переменными

Данное окно содержит локальные переменные, с помощью него можно отследить значения, которые хранятся в этих переменных.

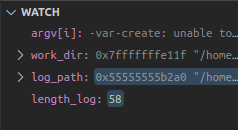


Рис. 3.6. Окно с наблюдаемыми переменными

С помощью этого окна программист может следить за значениями конкретных переменных, чтобы проконтролировать, как изменяются их значения, и тем самым, возможно, обнаружить ошибку.

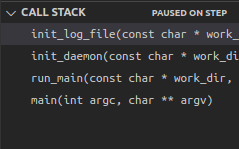


Рис. 3.7. Окно стэка вызовов функций

Стэк вызовов показывает, в какой последовательности вызывались функции.

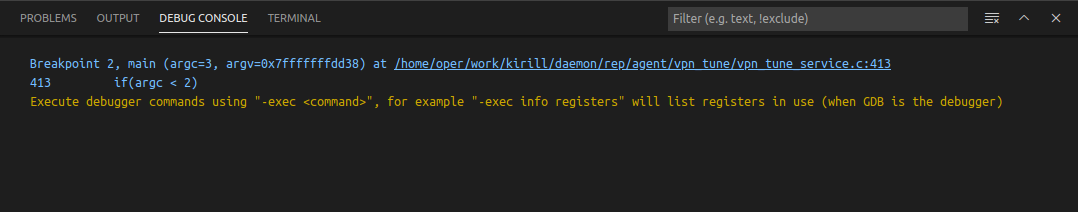


Рис. 3.8. Окно вывода

Все сообщения, которые должны выводиться на экран, будут показаны в этом окне. Также в этой области выводятся возникающие ошибки во время выполнения программы.

Благодаря такой отладке устранены ошибки, связанные с обработкой аргументов командной строки.

#### Отладка с помощью Valgrind

Valgrind – это система, находящаяся в открытом доступе, для отладки и профилирования программ, созданных для ОС Linux. С помощью набора инструментов Valgrind может автоматически обнаруживать множество ошибок управления памятью и потоковой передачи, делая программы более стабильными. Valgrind в настоящий момент предоставляет доступ к следующим инструментам: детектор ошибок памяти, детекторы ошибок потоков, профилировщик кэша и предсказания ветвления, профилировщик кучи.

Основными модулями в valgrind являются:

* memcheck – данный модуль является одним из основных при проверке корректности работы с памятью. Он обнаруживает ошибки при выделении, освобождении памяти. С помощью данного модуля можно найти и устранить утечки памяти, которые могут возникать в течении написания кода программы;
* cachegrind – данный модуль отвечает за профилирования кэша, то есть собирает статистку попадания (hit) и не попадания (miss) в кэш. Также этот модуль выдает статистику предсказаний ветвлений;
* callgrind – данный модуль в своей работе схож с предыдущем, отличие заключается в том, что callgrind анализирует стэк вызовов функций;
* massif – данный модуль занимается отслеживанием выделением памяти из кучи, он выдает информацию о том, какие части программы большего всего обращаются к кучи;
* helgrind – данный модуль отвечает за проверку обращений потоков к ячейкам памяти, он используется при написании программ с помощью организации многопоточности с использованием pthreads.

Работа с valgrind происходит исключительно в терминале при помощи заданий различных опций, также для использования данной технологии не нужно предварительно настраивать и подготавливать программу, но для получения полной информации о работе программы необходимо выполнить сборку с отладочной информации при помощи флагов –g.

Пример использования valgrind с отладкой ПМ МСИ:

valgrind -s --leak-check=full ./vpn\_tune\_service -f /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/

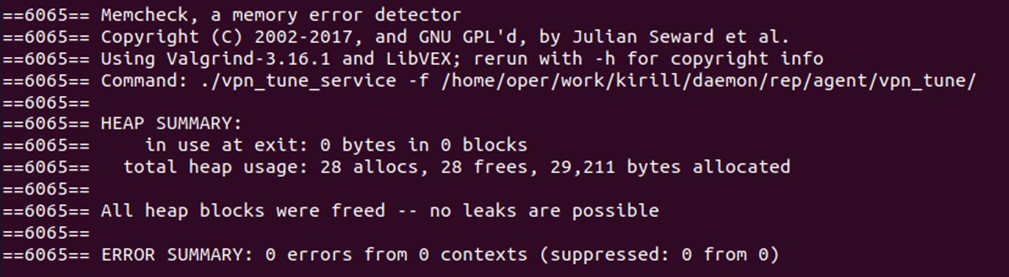


Рис. 3.9. Пример использования valgrind

В результате можно увидеть, что в программе отсутствуют утечки памяти и ошибки при выделении и использовании ее.

Если при запуске valgrind ничего не указывать дополнительно, то запуститься модуль memcheck. Для того чтобы использовать другие модули необходимо указать нужный модуль с помощью опции --tool, написав после знака равно имя модуля:

valgrind --tool=callgrind ./vpn\_tune\_service -f /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/

Если при отладке программы, в основном, используются только ряд определенных модулей, то их можно указать в специальном конфигурационном файле ~/.valgrindrc.

Существует ряд основных опций, которые используют наиболее часто при отладке программ:

* -h / --help – с помощью данного флага можно получить информацию о других флагах и опциях;
* -q / --quiet – при использовании данного флага на экран будут выводиться только сообщения об ошибках, все остальные сообщения показываться не будут;
* -v / --verbose –данная опция позволяет получить расширенную информацию об отлаживаемой программе;
* --log-file – при помощи данной опции можно задать путь к файлу, в котором будут собраны все сообщения, появляющиеся во время отладки valgrind;
* --track-fds – данная опция позволяет получить информацию об открытых дескрипторах;
* --xml – при указании данной опции на экран будет выведена информация об ошибках в формате XML;
* --time-stamp – эта опция позволяет отследить время возникновения того или иного сообщения об ошибке. Время отсчитывается от старта программы;
* --leak-check – с помощью этой опции можно отследить утечки памяти в программе.

Произведен разбор работы valgrind в случае наличия различных ошибок в программе.

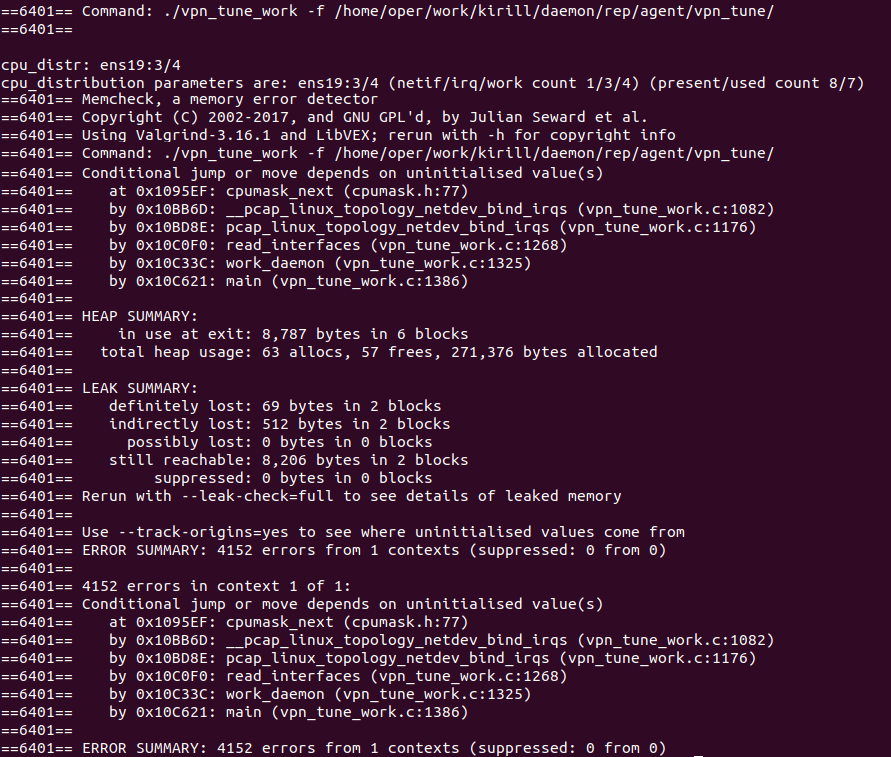


Рис. 3.10. Ошибка использования неинициализированной переменной

Такая ошибка связана с тем, что память выделяется динамически и перед непосредственным использованием переменной ей не задается никакое значение.

Благодаря опции --track-origins=yes можно узнать точную строку, в которой была объявлена переменная, которой не задали начальное значение.

Представлен пример обнаружения утечек памяти в программе при помощи опции --leack-check=full. Утечка памяти – это неконтролируемое уменьшение свободной оперативной или виртуальной памяти компьютера. Причиной утечек становятся ошибки в программном коде, а именно не освобождение памяти, выделенной динамически.

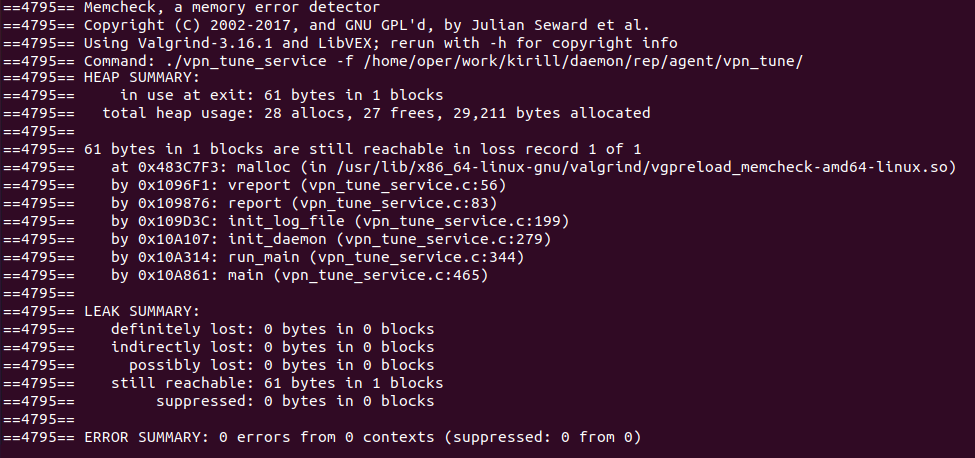


Рис. 3.11. Пример утечки памяти

С помощью valgrind устранены ошибки связанные с утечкой памяти и другими ошибками в использовании памяти.

## Анализ средств тестирования

Тестирование программного обеспечения – процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.

В тестировании есть две фундаментальные идеи:

* позитивное тестирование – это тестирование, когда рассматривается работоспособность программы в некоторых заданных условиях с учетом полной работы ПО, то есть программа отрабатывает так, как и было описано в требованиях;
* негативное тестирование – это тестирование, при котором выявляются все условия и наборы входных данных, при которых программа работает некорректно. При этом негативное тестирование является более полезным для оценки качества разработанного ПО, так как отражает все недочеты и ошибки в работе программы, которые необходимо устранять.

Целью тестирования является обеспечение должного качества продукта, а именно накопление информации, которая вернувшись разработчику, поможет ему в дальнейшем избежать подобных ошибок и улучшить качество ПО.

Перед началом тестирования ПМ МСИ изучены различные стратегии тестирования:

* стратегия поведенческого теста – это стратегия, которая основывается на технических требованиях. Например, тестирование всех функций описанных в требованиях, выполнение негативных тестов, которые вытекают из составленных требований. Такое тестирование также называется тестирование черным ящиком;
* стратегия структурного теста – это стратегия определяется структурой тестируемого объекта. Например, исполнения каждой функции объекта, прохода по всем предусмотренным ветвям алгоритма. Такое тестирование называется тестирование белым ящиком;
* стратегия гибридного теста – это стратегия является сочетанием поведенческой и структурной стратегий. Она сочетает в себе как тестирование непосредственно структуры объекта, так и тестирование поведения объекта в различных условиях.

Стратегия является эффективной, если тесты, входящие в данную стратегию, с большой долей вероятности обнаружат ошибки объекта.

Рассмотрены достоинства и недостатки каждой стратегии:

* метод черного ящика:
  + достоинства:
    - человек, который будет заниматься тестированием объекта данным способом, не обязательно обладать обширными знаниями в программировании;
    - поведение программного продукта исследуется в реальных пользовательских сценариях;
    - тест-кейсы можно составлять уже на стадии формирования стабильных требований к ПО;
    - процесс создания кейсов позволяет найти недостатки в требованиях.
  + недостатки:
    - высокая вероятность не затронуть часть возможных вариантов работы продукта;
    - разработка грамотных кейсов требует качественной документации;
    - тестирование может занимать большое количество времени.
* метод белого ящика:
  + достоинства:
    - автоматизация тестов и их использование со старта разработки проекта и на всем ее протяжении;
    - четко определенная система метрик и оценок;
    - покрытие всего кода программы.
  + недостатки:
    - использование данного метода может быть трудоемким, так как необходимо тестами стараться покрывать весь функционал модуля;
    - для тестирования данным методом необходимо обладать достаточными знаниями в области программирования;
    - при тестировании не учитывается влияние реальной среды использования;
    - большое количество времени необходимо для проведения тестирования большой информационной системы.
* Метод серого ящика содержит в себе достоинства и недостатки черного и белого ящика.

В качестве стратегии для тестирования ПМ МСИ выбрана стратегия поведенческого тестирования.

## Тестирование ПМ МСИ методом черного ящика

Все тестирование начинается с составления специализированных тест-кейсов. Все тест-кейсы разделяются на положительные и негативные сценарии.

Сначала тестирование ПМ МСИ будет проведено с положительными сценариями с целью протестировать режимы работы модуля.

Для проведения тестирования составлена таблица с описанием кейса, ожидаемым результатом, описанием набора входных данных и действительным результатом, полученным после выполнения сценария. При написании тестов необходимо стараться учесть максимально возможное количество различных вариантов использования продукта, чтобы убедиться в корректности работы ПО.

Табл. 3.1.

Сценарии положительных тестов для тестирования режимом работы модуля

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Описание сценария | Входные данные | Ожидаемый результат | Действи-  тельный результат | Вывод |
| Запуск сервиса в интерактивном режиме путем ввода соответствующего флага (-f) | ./vpn\_tune\_service -f /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Вся сообщения, которые генерирует программа, будут выведены на экран | Все сообщения выведены на экран | Тест пройден |
| Запуск сервиса в фоновом режиме | ./vpn\_tune\_service /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Все сообщения, которые генерирует программа, будут выведены в лог-файл | Все сообщения выведены в лог-файл | Тест пройден |
| Запуск сервиса в «сухом» режиме путем ввода соответствующего флага (-d) | ./vpn\_tune\_service  -f -d /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Вся сообщения, которые генерирует программа, будут выведены на экран и программа не изменит файлы smp\_affinity | Все сообщения выведены на экран, файлы smp\_affinity не изменены | Тест пройден |
| Запуска сервиса с указанием считывания информации о прерываниях из специального файла | ./vpn\_tune\_service  -f -d -t  work/daemon/rep/agent/vpn\_tune/  work/daemon/rep/agent/vpn\_tune/ | Данные о прерываниях будут получены из специального файла | Данные считаны из файла | Тест пройден |
| Проверка изменения файла smp\_affinity | ./vpn\_tune\_service -f /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Измененный файл smp\_affinity, в котором сохранена маска | В файле smp\_affinity находится новая маска | Тест пройден |

Подробный разбор каждого из тестов:

1. Запуск сервиса в интерактивном режиме путем ввода соответствующего флага   
   (-f). Суть теста заключается в том, чтобы проследить, выводиться ли на экран пользователю промежуточная и итоговая информация. Входными данными для этого теста является командная строка с определенными аргументами (название исполняемого файла, путь к конфигурационным файлам, флаг интерактивного режима). Выходными данными является все сообщения, создаваемые программой. Ожидаемым результатом выступает информация, которая будет распечатана в терминал.



Рис. 3.12. Выполнение теста для проверки интерактивного режима работы

При выполнении теста получен ожидаемый результат – сообщения распечатаны в терминал, следовательно, тест-кейс пройден.

1. Запуск сервиса в фоновом режиме. Тест заключается в том, что проверяется запись сообщений в лог-файл. Входными данными является командная строка. Выходными данными является сформированный лог-файл. В качестве результата ожидается получить сформированный лог-файл и отсутствие сообщений в консоли.

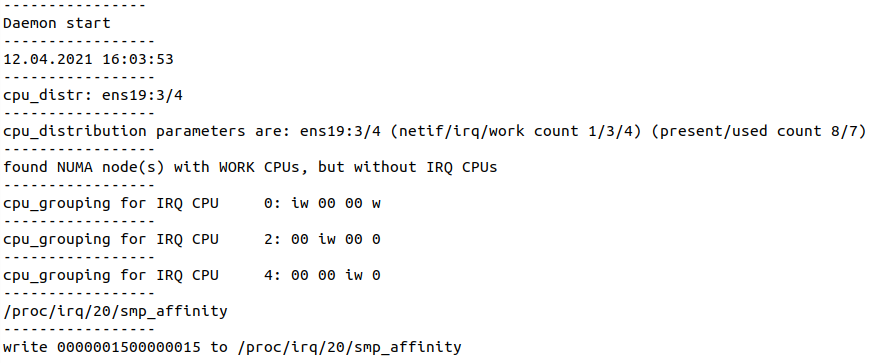


Рис. 3.13. Сформированный лог-файл после выполнения теста

1. Запуск сервиса в «сухом» режиме путем ввода соответствующего флага (-d). В данном тесте отслеживается вывод информации на экран пользователя и то, что программа не вносит изменений в файлы smp\_affinity. Проверка изменений файла smp\_affinity осуществляется проверкой значения, которое хранилось в файле до запуска модуля и после работы модуля. Входными данные также является командная строка, а выходными являются сообщения в терминале пользователя и неизмененный файл smp\_affinity. Результат выполнения теста представлен на рисунке.

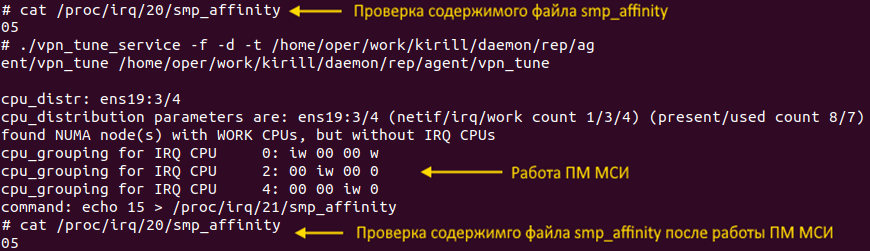


Рис. 3.14. Выполнение теста при запуске модуля в «сухом» режиме

Как видно из рисунка значение файла smp\_affinity не изменилось, следовательно, тест пройден.

1. Запуска сервиса с указанием считывания информации о прерываниях из специального файла. Данный сценарий показывает, что модуль может работать не только с потоком /proc/interrupts, но и с данными взятыми из специального файла. Входными данными является командная строка и файл с прерываниями. Выходными данными является информация, выводимая в терминал и сформированный лог-файл. Проверкой того пройден тест или нет служит номер IRQ прерывания, по которому осуществилась бы запись маски. В случае если чтение данных происходит из файла, то пользователь увидит IRQ под номером 21, так как именно такой номер имеет интересующий пользователя интерфейс в данном файле. Строка, которая парсится в этом случае выглядит следующем образом:



Результат выполнения теста:

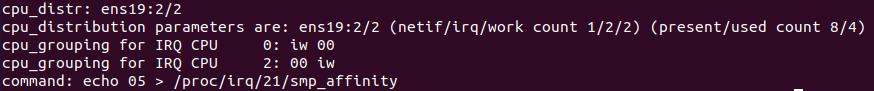


Рис. 3.15. Выполнение теста при использовании файла с прерываниями

Как видно из рисунка пользователь получил IRQ равное 21, следовательно, тест пройден.

1. Проверка изменения файла smp\_affinity. Один из самых главных тестов-кейсов, так как данная функция является основой и целью работы ПМ МСИ. Входными данными является командная строка. Выходными данными является измененный файл smp\_affinity. Проверка записи в файл осуществляется путем анализа содержимого файла до работы модуля и после. Проверяется это с помощью команды «cat /proc/irq/20/smp\_affinity».

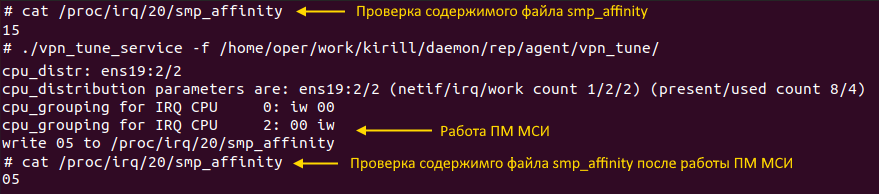


Рис. 3.16. Выполнение теста для проверки изменения файла smp\_affinity

Как видно из рисунка значение файла после выполнения программы поменялось на то, которое сформировала программа, следовательно, тест пройден.

После того как проведены положительные тесты, разработаны и выполнены негативные тесты, сценарии которых приведены в таблице.

Табл. 3.2.

Сценарии негативных тестов для тестирования режимом работы модуля

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Описание сценария | Входные данные | Ожидаемый результат | Действи-  тельный результат | Вывод |
| Запуск сервиса без указания пути к конфигурационным файлам | ./vpn\_tune\_service | Вывод сообщения об ошибке | Выведено сообщение об ошибке | Тест пройден |

Продолжение табл. 3.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Запуск сервиса с флагом -d без указания флага -f | ./vpn\_tune\_service –d /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Вывод сообщения об ошибке | Выведено сообщение об ошибке | Тест пройден |
| Запуск сервиса с указанием флага -t без указания пути к файлу с прерываниями | ./vpn\_tune\_service  -f -d -t /home/oper/work/kirill/daemon/rep/agent/vpn\_tune/ | Вывод сообщения об ошибке | Выведено сообщение об ошибке | Тест пройден |

Подробный разбор каждого из тестов:

1. Запуск сервиса без указания пути к конфигурационным файлам. В данном тесте проверяется обработка ошибки запуска сервиса, так как путь к конфигурационным файлам является обязательным параметром. Входными данными является командная строка, содержащая название исполняемого файла. Выходными данными является сообщение об ошибке. Оператору на экран было выведено информационное сообщение, которое содержит описание ошибки, также выведено правило запуска модуля.

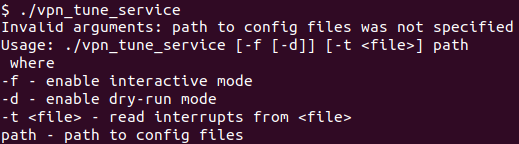


Рис. 3.17. Выполнение теста для проверки обработки ошибки не указания пути к конфигурационным файлам

Как видно из рисунка тест пройден, пользователю выведено на экран сообщение о том, что необходимо указывать обязательно путь, также приведен пример запуска с флагами, отмеченными квадратными скобками, так как использование флагов не является обязательным. Приведена расшифровка каждого из флагов.

1. Запуск сервиса с флагом -d без указания флага -f. Данный тест проверяет обработку ошибки, когда используется флаг -d, без указания флага интерактивного режима (-f), так как при использовании «сухого» режима обязательно использовать флаг интерактивного режима. Входными данными является командная строка, содержащая имя исполняемого файла, флаг -d и путь к файлам. Выходными данными является сообщение об ошибке, которое содержит описание ее и пример запуска сервиса.

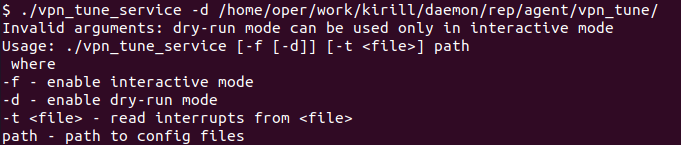


Рис. 3.18. Выполнение теста запуска сервиса с флагом -d без указания флага -f

Как видно из рисунка было выведено сообщение об ошибке, следовательно, тест пройден.

1. Запуск сервиса с указанием флага -t без указания пути к файлу с прерываниями. В данном сценарии проверяется обработка ошибки, когда пользователь не указывает путь до файла с прерываниями. Входные данные – это командная строка, содержащая имя исполняемого файла, флаг -t и путь к конфигурационным файлам. Выходными данными является сообщение об ошибке, которое содержит описание ее и пример запуска сервиса.

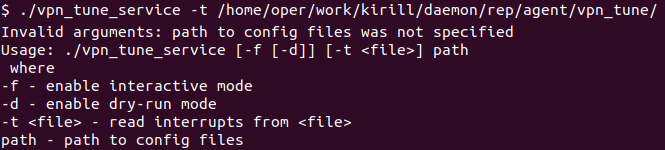


Рис. 3.19. Выполнение теста без указания пути к файлу с прерываниями

Как видно из рисунка ошибка обработана корректно, следовательно, тест пройден.

Также было проведено тестирование обработки интервала работы модуля. Разработанные тест-кейсы разделяются на несколько типов:

* проверка граничных значений;
* проверка из положительного диапазона;
* проверка не валидных значений:
  + пустая строка;
  + буквенные символы;
  + отрицательные значения.

Так как тестирование происходит в рамках метода черного ящика невозможно точно протестировать верхнюю границу допустимого диапазона, потому что реализация при данном методе скрыта. Поэтому будут выбраны несколько типовых значений для того, чтобы методом подбора определить верхнюю границу:

* максимальное значение для типа INT = 2 147 483 647;
* максимальное значение для типа UINT (unsigned) = 4 294 967 295;
* максимальное значение для типа LLONG = 9 223 372 036 854 775 807;
* максимальное значение для типа ULLONG = 18 446 744 073 709 551 615.

Если будет найдено граничное значение до тестирования значения ULLONG, то оставшиеся значения протестированы не будут. Тест-кейсы приведены в таблице.

Табл. 3.3.

Сценарии тестирования обработки интервалов работы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Описание сценария | Входные данные | Ожидаемый результат | Действи-  тельный результат | Вывод |
| Интервал работы = 10 | Interval 10 | Вызов подпрограммы будет происходить каждые 10с | Прочитано верное значение равное 10с | Тест пройден |
| Интервал работы = 1 | Interval 1 | Вызов подпрограммы будет происходить каждую 1с | Прочитано верное значение равное 1с | Тест пройден |
| Интервал работы = 2147483647 | Interval 2147483647 | Вызов подпрограммы будет происходить каждые 2147483647с | Прочитано верное значение равное 2147483647с | Тест пройден |
| Интервал работы = 4294967295 | Interval 4294967295 | Вызов подпрограммы будет происходить каждые 4294967295с | Выведена ошибка (слишком большое значение) | Тест не пройден |

Продолжение табл. 3.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Интервал работы = 0 | Interval 0 | Вывод сообщения об ошибке (некорректное значение интервала) | Выведено сообщение об ошибке | Тест пройден |
| Интервал работы = -1 | Interval -1 | Вывод сообщения об ошибке (некорректное значение интервала) | Выведено сообщение об ошибке | Тест пройден |
| Интервал работы = «»  (пустая строка) | Interval | Вывод сообщения об ошибке (некорректное значение интервала) | Выведено сообщение об ошибке | Тест пройден |
| Интервал работы = «num» | Interval num | Вывод сообщения об ошибке (некорректное значение интервала) | Выведено сообщение об ошибке | Тест пройден |

Подробный разбор каждого из сценария:

1. Интервал работы равен 10 секундам. Данный тест проверяет значение из корректного диапазона. Входными данными является строка «Interval 10». Выходными данными является считанное значение из файла. Проверка правильности считывания значения осуществляется по средствам наличия или отсутствия ошибки в терминале.

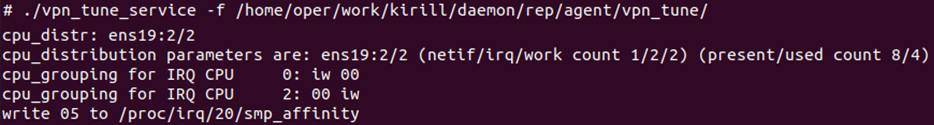


Рис. 3.20. Выполнение теста при значении интервала равное 10

Как видно из рисунка программа выполнила одну итерацию, следовательно, значение прочитано верно, поэтому тест считается пройденным.

1. Интервал работы равен 1 секунде. В данном тесте-кейсе проверяется одно из граничных значений. В данном случае входными данными является строка «Interval 1», выходными данными является значение переменной.

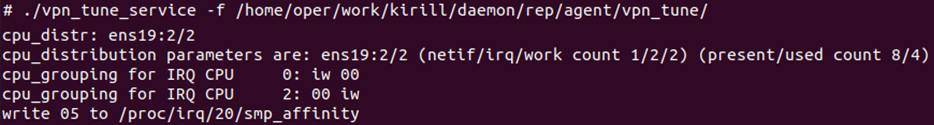


Рис. 3.21. Выполнение теста при значении интервала равное 1

В терминале не пояивлось сообщение об ошибке, значит тест пройден.

1. Интервал работы равен 2147483647 секунд. Данное значение используется в качестве предположительного значения правой (верхней) границы допустимых значений интервала. Входными данными является строка «Interval 2147483647», выходными является значение, прочитанное из файла и сохраненное в переменной num. Выполнив тест, получен ожидаемый результат, а именно была выполнена одна итерация программы, следовательно, тест пройден. Так как данный тест пройден, необходимо протестировать следующее значение, для определения правой границы.

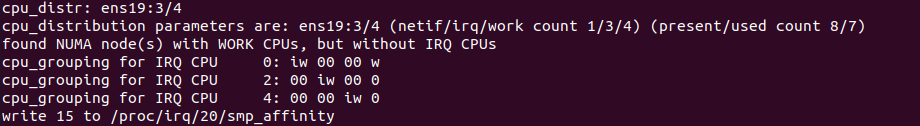


Рис. 3.22. Выполнение теста при значении интервала равное 2147483647

1. Интервал работы равен 4294967295 секунд. Так как предыдущий тест пройден, этот тест должен проверить следующее значение – ULONG\_MAX. Входными данными является строка «Interval 4294967295», выходными является значение, прочитанное из файла и сохраненное в переменной num.



Рис. 3.23. Выполнение теста при значении интервала равное 4294967295

В терминале появилась ошибка, которая свидетельствует о том, что значение 4294967295 является не валидным. Тест не пройден. Следующие сценарии для проверки верхней границы проводиться не будут, так как было выяснено, что верхней границей является значение INT\_MAX.

1. Интервал работы равен 0 секунд. В данном случае ожидаемым результатом является появление сообщения об ошибке в терминале. Входными данными является строка «Interval 0». Выходными данными является значение, прочитанное из файла и сохраненное в переменной num.



Рис. 3.24. Выполнение теста при значении интервала равное 0

После выполнения теста была выведена ошибка, что совпадает с ожидаемым результатом, следовательно, тест-кейс пройден.

1. Интервал работы равен -1 секунде. Это значение используется в качестве проверки работы программы в различных внештатных ситуаций, которые могут происходить в период эксплуатации продукта. Ожидаемым результатом является сообщение об ошибке.



Рис. 3.25. Выполнение теста при значении интервала равное -1

На экран было выведено сообщение об ошибке, значит, тест считается пройденным.

1. Интервал работы равен пустой строке. Данная проверка показывает, как обрабатывает программа не валидные значения. Предполагается, что программа выдаст ошибку.



Рис. 3.26. Выполнение теста при значении интервала равное пустой строке

В терминал было выдано сообщение об ошибке в задании интервала времени, следовательно, тест пройден.

1. Интервал работы равен строке. Проверяется реакция программы на то, что в качестве значения интервала работы будет подано символьное выражение (например, будет подана строка «Interval num»). Программа должна выдать ошибку.



Рис. 3.27. Выполнение теста при значении интервала равное строке

Программа обработала строковое значение и выдала ошибку, тест был пройден.

## Выводы по технологическому разделу

В данном разделе приведен процесс сборки ПМ МСИ и анализ отладки ПМ МСИ с помощью средств отладчика GDB и средства Valgrind. Подробно рассмотрены экранные формы процесса отладки с помощью встроенного отладчика в среду Visual Studio Code и экранные форма средств Valgrind.

Также подробно рассмотрен процесс тестирования модуля. Выбран метод, с помощью которого проводилось тестирование работоспособности ПО, разработаны тест-кейсы разного типа для проверки разного функционала. Приведены результаты выполнения тестов.

# ЗАКЛЮЧЕНИЕ

В рамках данной ВКР поставлена цель - повышение точности корректировки масок прерываний сетевых интерфейсов. Исследована предметная область, найдены недостатки в существующих решениях, составлены требования к ПМ МСИ, по которым велась дальнейшая разработка. Также разработаны схемы алгоритма и схемы данных, на основе их осуществлялась разработка структур входных и выходных данных. Разработано руководство оператора.

Работа завершена и ее результатом является отлаженная и протестированная программная реализация ПМ МСИ, применяемая на предприятии ООО «С-Терра СиЭсПи».

# СПИСОК ЛИТЕРАТУРЫ

1. Гагарина Л.Г., Касимов Р.А., Коваленко Д.Г., Федотова Е.Л, Чжо Зо Е, Черников Б.В. Методические указания по подготовке выпускной квалификационной работы по направлению подготовки бакалавров 09.03.04 «Программная инженерия»/ Под редакцией Б.В. Черникова; М., МИЭТ, 2016 г., 20 с.
2. Беклемишев Д. Н., Орлов А. Н., Переверзев А. Л., Попов М. Г., Горячев А. В., Кононова А. И. Микропроцессорные средства и системы: курс лекций / Под ред. Ю. В. Савченко. — М.: МИЭТ, 2013. — 288 с.
3. Борис, Бейзер Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Бейзер Борис. — 1. — Санкт-Петербург : Питер, 2004. — 318 c.
4. Герберт, Шилдт Java: руководство для начинающих / Шилдт Герберт. — 7. — Санкт-Петербург : Диалектика, 2019.
5. Клинтон, Дэвид Linux в действии / Дэвид Клинтон. — 1. — : Издательский Дом ПИТЕР, 2019. — 416 c.
6. Куликов, С. Тестирование программного обеспечения. Базовый курс. / С. Куликов. — 3. — : EPAM Systems, 2021. — 298 c.
7. C/C+. Программирование на языке высокого уровня : Учебник для вузов / Павловская Т.А.. - СПб. : Питер, 2005, 2001
8. Страуструп, Бьярне Программирование. Принципы и практика с использованием C++ / Бьярне Страуструп. — 2. — : Вильямс, 2018. — 1328 c.
9. Atom // Atom URL: https://atom.io/
10. Eclipse // Eclipse URL: https://www.eclipse.org/
11. Intel // Базовый драйвер для Linux \* для сетевых гигабитных Ethernet/адаптеров Intel® URL: https://www.intel.ru/content/www/ru/ru/support/articles/000005480/network-and-io/ethernet-products.html
12. Microsoft // Visual Studio URL: http://msdn.microsoft.com/ru-ru/vcsharp/default.aspx
13. Microsoft // Visual Studio Code URL: https://code.visualstudio.com/
14. Oracle // Java URL: http://www.oracle.com/technetwork/java/index.html
15. The Linux Kernel // What is NUMA? URL: https://www.kernel.org/doc/html/v4.18/vm/numa.html
16. The Linux Kernel // SMP IRQ affinity URL: https://www.kernel.org/doc/Documentation/IRQ-affinity.txt
17. s-terra портал документации // CPU\_DISTRIBUTION URL: http://doc.s-terra.ru/rh\_output/4.3/Gate/output/index.htm
18. Sublime Text 3 // Sublime Text 3 URL: https://www.sublimetext.com/docs/3/linux\_repositories.html
19. Ubuntu manuals // Writing and packaging system daemons URL: http://manpages.ubuntu.com/manpages/bionic/man7/daemon.7.html
20. НОУ ИНТУИТ // Лекция 3: Базовые понятия ОС Windows URL: https://intuit.ru/studies/courses/962/217/lecture/5589 (дата обращения: 09.03.2021).
21. What is GDB?. // GDB: The GNU Project Debugger URL: https://www.gnu.org/software/gdb/https://valgrind.org/
22. Options to Request or Suppress Warnings. // GDB. — URL: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
23. NUMA(7). — Текст : электронный // man7 — URL: https://man7.org/linux/man-pages/man7/numa.7.html
24. Сетевая подсистема Linux. // НОУ ИНТУИТ — URL: https://intuit.ru/studies/courses/681/537/lecture/12106
25. lscpu - display information about the CPU architecture — // die.net — URL: https://linux.die.net/man/1/lscpu
26. ФАЙЛОВАЯ СИСТЕМА PROC В LINUX — // losst — URL: https://losst.ru/fajlovaya-sistema-proc-v-linux
27. getopt. — // openNET — URL: https://www.opennet.ru/man.shtml?topic=getopt&category=3&russian=0
28. GCC, the GNU Compiler Collection. — // GCC — URL: https://gcc.gnu.org/
29. Official Ubuntu Documentation. — // Ubuntu — URL: https://help.ubuntu.com/?\_ga=2.167278590.1795639592.1621588680-1331352559.1615310715
30. git documentation. — // git — URL: https://git-scm.com/
31. SMP IRQ affinity. — // kernel — URL: https://www.kernel.org/doc/Documentation/IRQ-affinity.txt
32. irqbalance. — // GitHub irqbalance — URL: https://github.com/Irqbalance/irqbalance
33. sprintf. — // Программирование на C и C++ — URL: http://www.c-cpp.ru/content/sprintf
34. create daemon in Linux. — // OpenNET — URL: https://www.opennet.ru/base/dev/daemon\_example.txt.html
35. daemon — // man7 — URL: https://man7.org/linux/man-pages/man7/daemon.7.html
36. Аргументы функции MAIN(). — // Программирование на C и C++ — URL: http://www.c-cpp.ru/books/argumenty-funkcii-main
37. аргументы функции и командной строки. — // Microsoft Build — URL: https://docs.microsoft.com/ru-ru/cpp/cpp/main-function-command-line-args?view=msvc-160
38. memcpy, wmemcpy. — // Microsoft Build — URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-160
39. vprintf Functions. — // Microsoft Build — URL: https://docs.microsoft.com/en-us/cpp/c-runtime-library/vprintf-functions?view=msvc-160
40. Process Identification. — // GNU — URL: https://www.gnu.org/software/libc/manual/html\_node/Process-Identification.html
41. getpid. — // Программирование на C и C++ — URL: http://www.c-cpp.ru/content/getpid
42. Функция popen(). — // OpenNET — URL: https://www.opennet.ru/docs/RUS/linux\_parallel/node16.html
43. Создание процессов с помощью вызова fork(). — // OpenNET — URL: https://www.opennet.ru/docs/RUS/linux\_parallel/node7.html
44. goto и помеченные операторы (C). — // Microsoft Build — URL: https://docs.microsoft.com/ru-ru/cpp/c-language/goto-and-labeled-statements-c?view=msvc-160
45. Основные классы современных параллельных компьютеров. — // Parallel — URL: https://parallel.ru/computers/classes.html
46. int system(const char \*str). — // Программирование на C и C++ — URL: http://www.c-cpp.ru/content/system
47. МЕЖГОСУДАРСТВЕННЫЙ СТАНДАРТ. Единая система программной документации СХЕМЫ АЛГОРИТМОВ, ПРОГРАММ, ДАННЫХ И СИСТЕМ. Обозначения условные и правила выполнения. — // Электронный фонд правовых и нормативно- технических документов — URL: https://docs.cntd.ru/document/9041994
48. strtoul, \_strtoul\_l, wcstoul, \_wcstoul\_l. — // Microsoft Build — URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/strtoul-strtoul-l-wcstoul-wcstoul-l?view=msvc-160
49. strftime. — // cplusplus — URL: https://www.cplusplus.com/reference/ctime/strftime/
50. Работа с файлами. — // prog-c — URL: https://prog-cpp.ru/c-files/
51. stdin, stdout, stderr. — // Microsoft Build — URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/stdin-stdout-stderr?view=msvc-160
52. Понятие о сигналах. — // OpenNET — URL: https://www.opennet.ru/docs/RUS/linux\_parallel/node10.html
53. Сигналы, группы, сеансы. — // Параллельные вычисления в УрО РАН — URL: https://parallel.uran.ru/book/export/html/399
54. Утверждаю
55. Директор института СПИНТех
56. НИУ МИЭТ
57. Проф. \_\_\_\_\_\_\_\_\_\_/Гагарина Л.Г./
58. «\_\_\_» \_\_\_\_\_\_\_\_\_ 2021 г.

Техническое задание на разработку

«Программный модуль для мониторинга и корректировки прерываний сетевых интерфейсов»

(шифр ПМ МСИ)

1. Направление подготовки – 09.03.04
2. Квалификация – бакалавр

Руководитель выпускной работы:

К.п.н., доцент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Федотова Е.Л./

2. Исполнитель:
3. Студент гр. ПИН-41 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Поселенов К.В./

Москва 2021

1. Введение

В компании ООО «С-Терра СиЭсПи» используются ЭВМ, в которые устанавливаются многоядерные процессоры, где количество ядер, как правило, больше восьми. Через такие ЭВМ проходит огромное количество сетевых пакетов данных. Поэтому залогом быстрой и эффективной работы ЭВМ является грамотное распределение нагрузки по доступным ядрам процессора, для того чтобы время обработки одного пакета было минимальным. В связи с этим в системе применяется механизм прерываний.

Прерывание — это событие, генерируемое внешним (по отношению к процессору) устройством. В системе ЭВМ можно настраивать прерывания в зависимости от потребностей пользователя, это позволяет равномерно распределить работу по ядрам процессора. Оператор ЭВМ может самостоятельно определить, какие конкретно ядра будут обрабатывать то или иное прерывание, что позволит ему скорректировать работу всей ЭВМ. Для удобного изменения масок прерываний было создано решение внутри компании, которое представлено в виде драйвера, работающего на уровне ядра. Проанализировав уже существующие и аналогичные решения, было принято решение о разработке программного модуля, работающего циклически в фоновом режиме с заданным интервалом времени.

1. Основания для разработки
   1. Основание для разработки

– Задание на выпускную квалификационную работу;

– Решение директора института СПИНТех.

* 1. Наименование разработки

«Разработка программного модуля мониторинга и корректировки прерываний сетевых интерфейсов». Программному модулю, разрабатываемому по настоящему ТЗ, присваивается шифр «ПМ МСИ» (Далее в тексте - ПМ).

* 1. Исполнитель

Исполнителем является студент группы ПИН-41 НИУ МИЭТ Поселенов Кирилл Владимирович.

1. Назначение разработки

Данный ПМ создается с целью упрощения контроля и выставления прерываний для необходимых сетевых интерфейсов, при помощи задания нужной информации в специальные конфигурационные файлы.

1. Технические требования
   1. Требования к функциональным характеристикам
      1. Программный модуль должен обеспечивать возможность выполнения следующих функций:

* считывание входных данных:
  + считывание интервала работы модуля из файла;
  + считывание данных о сетевых интерфейсах из файла:
    - считывание названия;
    - считывание количества обрабатывающих ядер;
  + считывание входных аргументов из командной строки:
    - задание режимов работы:
      * интерактивный режим работы;
      * «сухой» режим работы;
      * режим работы, когда считывание данных о прерываниях задается из файла, а не из операционной системы;
      * путь до файла с прерываниями в случае, если задан флаг –t.
    - задания пути к конфигурационным файлам;
* формирование масок ядер процессора в каждой новой итерации работы модуля;
* формирование массива нума-нод процессора;
* формирование маски прерывания для сетевого интерфейса;
* установление маски прерывания для сетевого интерфейса путем записи в файл smp\_affinity;
* наличие управляющего сервиса;
* наличие рабочей части программы, которая будет формировать и записывать маску в файл smp\_affinity;
* работа в циклическом режиме с определенным интервалом работы, задаваемым в конфигурационном файле;
* возможность работы в фоновом режиме;
* возможность работы в интерактивном режиме для осуществления отладки и тестирования;
* запись всех промежуточных и итоговых результатов в лог-файлы.  
  + 1. Организация входных и выходных данных

Входные данные:

В качестве входных данных для ПМ выступают данные в конфигурационных файлах, которые несут информацию о прерываниях сетевых интерфейсов, а также информация из командной строки, в которой заключены:

* Режим работы;
* Путь к конфигурационным файлам.

Выходные данные:

Выходными данными являются:

* Лог-файл, в котором хранятся все промежуточные и итоговые результаты одной итерации работы ПМ;
* Вывод вспомогательной информации на экран пользователю при использовании «интерактивного» и «сухого» режима работы.
  1. Требования к надежности

Работа ПМ не должна приводить к сбоям операционной системы. ПМ должен работать с входными данными, предусмотренными техническими требованиями в соответствии с алгоритмом функционирования, обрабатывать и сообщать об ошибках при неверном задании входных данных и прочих внештатных ситуациях.

* 1. Условия эксплуатации

Оператор, работающий с ПМ, должен быть ознакомлен с документацией компании ООО «С-Терра СиЭсПи» по заданию формата прерываний в конфигурационном файле. Также оператор должен обладать навыками работы в ОС Linux.

* 1. Требования к составу и параметрам технических средств

В состав технических средств должен входить ПК на базе ОС Linux, включающий в себя:

* процессор 2.0 ГГц или выше;
* количество ядер процессора 4 или выше;
* ОЗУ 2 ГБ;
* жесткий диск HDD, 5400 об/мин;
* объем доступного пространства на жестком диске не менее 5 ГБ;
* видеоадаптер с поддержкой DirectX 9 и разрешения экрана 1024х768;
* устройства ввода/вывода: мышь, клавиатура, монитор.
  1. Требования к информационной и программной совместимости

Базовый язык программирования – С, среда разработки Visual Studio Code. ПМ должен работать под ОС Linux.

* 1. Специальные требования

Специальных требований к характеристикам программы не предъявляется.

1. Требования к программной документации
   1. Требования к составу программной документации

В комплект документации должны входить: руководство пользователя.

* 1. Требования к оформлению документации

Программная документация должна быть разработана и оформлена в соответствии с ЕСПД.

1. Порядок контроля и приёмки

Контроль и приёмка разработки осуществляется на ЭВМ Заказчика на основе испытаний контрольно-отладочных примеров. При этом проверяется выполнение всех функций ПМ.

1. Стадии и этапы разработки

В течении периода с февраля 2021 года по июнь 2021 года должны быть проведены работы, указанные в Таблице 1.

Табл. 1.

График работ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *№* | *Наименование работы* | *Дата начала* | *Дата окончания* | *Форма отчетности* |
| *1* | *Постановка задачи* | *10.02.2021* | *11.02.2021* | *Эскиз слайда* |
| *2* | *Проведение предварительных НИР* | *12.02.2021* | *15.02.2021* | *Сравнительная таблица* |
| *3* | *Разработка Т3* | *16.02.2021* | *22.02.2021* | *Утвержденное ТЗ* |
| *4* | *Разработка структуры входных и выходных данных* | *23.02.2021* | *05.03.2021* | *Схема данных* |
| *5* | *Уточнение методов решения* | *06.03.2021* | *09.03.2021* | *Текстовый документ* |
| *6* | *Разработка общего алгоритма работы ПМ* | *12.03.2021* | *16.03.2021* | *Схема алгоритма* |

Продолжение табл. 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *7* | *Уточнение структуры входных и выходных данных* | *19.03.2021* | *21.03.2021* | *Уточненная схема данных* |
| *8* | *Разработка алгоритма работы ПМ* | *22.03.2021* | *25.03.2021* | *Алгоритм модуля* |
| *9* | *Программирование, отладка и тестирование* | *26.03.2021* | *01.05.2021* | *Тексты программ с комментариями* |
| *10* | *Разработка программных документов* | *04.05.2021* | *11.05.2021* | *Текстовые документы* |
| *11* | *Разработка программы и методики испытаний* | *12.05.2021* | *20.05.2021* | *Тест-кейс* |
| *12* | *Проведение испытаний* | *21.05.2021* | *22.05.2021* | *Результаты испытаний* |
| *13* | *Корректировка программы и документов по результатам испытаний* | *23.05.2021* | *24.05.2021* | *Скорректированные документы* |
| *15* | *Передача программы и документов для сопровождения* | *25.05.2021* | *06.06.2021* | *Рабочие документы* |

ПРИЛОЖЕНИЕ 2

ПРОГРАММНЫЙ МОДУЛЬ МОНИТОРИНГА И КОРРЕКТИРОВКИ ПРЕРЫВАНИЙ СЕТЕВЫХ ИНТЕРФЕЙСОВ (ПМ МСИ)

# РУКОВОДСТВО ОПЕРАТОРА

Москва, 2021

# АННОТАЦИЯ

В данном программном документе приведено руководство оператора по эксплуатации ПМ МСИ.

В разделе «Назначение и условия применения программы» указаны назначение и функции, выполняемые программным модулем, условия, необходимые для выполнения.

В разделе «Выполнение программы» представлена последовательность действий оператора для запуска ПМ МСИ, расписаны все параметры командной строки, приведены примеры запуска модуля.

В разделе «Сообщения» указаны тексты сообщений, выдаваемых программисту в ходе выполнения программы, описание их содержания и действия, которые необходимо предпринять по этим сообщениям.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД (ГОСТ 19.101-77, ГОСТ 19.103-77, ГОСТ 19.104-78, ГОСТ 19.105-78, ГОСТ 19.106-78, ГОСТ 19.505-79).

# СОДЕРЖАНИЕ

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ 89](#_Toc75293745)

[1. НАЗНАЧЕНИЕ И УСЛОВИЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ 90](#_Toc75293746)

[1.1. Функциональное назначение 90](#_Toc75293747)

[1.2. Эксплуатационное назначение 90](#_Toc75293748)

[1.3. Требования к аппаратным средствам 90](#_Toc75293749)

[1.4. Требования к программному обеспечению 91](#_Toc75293750)

[1.5. Требования к оператору 91](#_Toc75293751)

[2. ВЫПОЛНЕНИЕ ПРОГРАММЫ 92](#_Toc75293752)

[3. СООБЩЕНИЯ 94](#_Toc75293753)

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

CPU – central processing unit;

NUMA – none – uniform memory access;

IRQ – interrupt request;

ОС – операционная система.

# НАЗНАЧЕНИЕ И УСЛОВИЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ

## Функциональное назначение

ПМ МСИ обладает следующем функционалом:

* формирование масок online CPU;
* формирование массива numa node;
* парсинг аргументов командной строки для определения режима работы и пути к конфигурационным файлам;
* парсинг конфигурационных файлов для определения интервала работы модуля, распределения irq\_cores / working\_cores, имен сетевых интерфейсов, которые необходимо корректировать;
* формирование маски прерывания для каждого заданного сетевого интерфейса;
* определение номера прерывания IRQ по имени интерфейса;
* запись сформированной маски прерывания в соответствующий файл smp\_affinity;
* запись всей промежуточной и итоговой информации о работе модуля в лог-файл;
* получение информации о прерываниях из файла /proc/interrupts;
* возможность считывания информации о прерываниях из специального файла, подготовленным оператором;
* возможность работы как в фоновом режиме, так и в интерактивном.

## Эксплуатационное назначение

ПМ МСИ предназначен для мониторинга и корректировки масок прерываний сетевых интерфейсов.

## Требования к аппаратным средствам

Минимальный состав технических средств и их технические характеристики:

* процессор 2.0 ГГц или выше;
* количество ядер процессора 4 или выше;
* ОЗУ 2 ГБ;
* жесткий диск HDD, 5400 об/мин;
* объем доступного пространства на жестком диске не менее 5 ГБ;
* видеоадаптер с поддержкой DirectX 9 и разрешения экрана 1024х768;
* устройства ввода/вывода: мышь, клавиатура, монитор.

## Требования к программному обеспечению

ПМ МСИ работает в системах под управлением ОС Linux.

## Требования к оператору

Оператор, эксплуатирующий ПМ МСИ, должен знать документацию компании ООО «С-Терра СиЭсПи» по формированию cpu\_distribution, также должен уметь обращаться с терминалом ОС Linux.

# ВЫПОЛНЕНИЕ ПРОГРАММЫ

Запуск ПМ МСИ осуществляется в терминале ОС Linux, помощью вызова исполняемого файла. Пример запуска модуля:

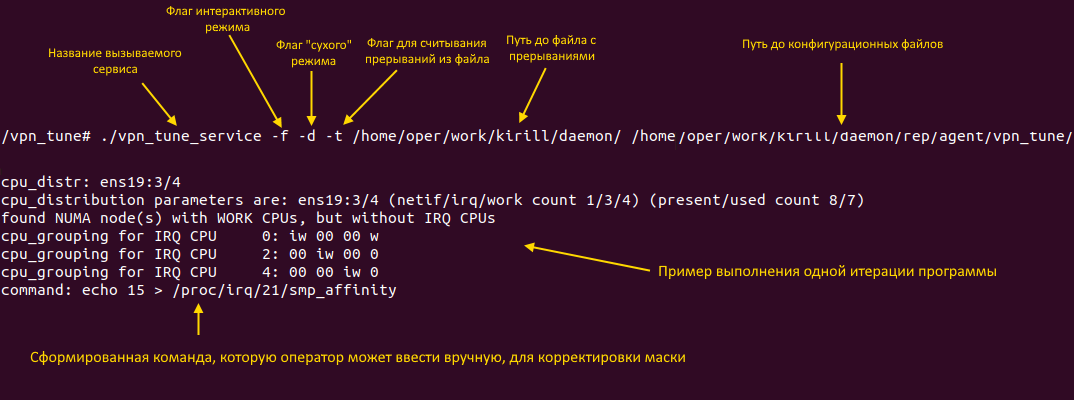


Рис. 2.1. Пример запуска ПМ МСИ с использованием флагов

У сервиса есть различные режимы работы, которые выбираются с помощью соответствующих флагов:

1. -f – флаг интерактивного режима работы. Все сообщения будут выводиться не только в специализированный лог-файл, но и на экран пользователя (не является обязательным параметром), программа не переходит в режим демона;
2. -t – флаг считывания данных о прерывании из заготовленного файла. Данный флаг позволяет отлаживать программу на тестовых данных, чтобы была возможность протестировать все варианты работы модуля перед его внедрением в реальную систему (не является обязательным параметром);
3. -d – флаг «сухого» режима работы. При данном режиме работы все сообщения поступают на экран пользователя, а также модуль не вносит никаких изменений в работу системы (не является обязательным параметром).

Обязательным параметром при запуске модуля является только путь до конфигурационных файлов.

Также для работы модуля необходимо предварительно создать конфигурационный файл «config.txt», в котором содержится информация об интервале работы модуля и список имен сетевых интерфейсов, за которыми модулю необходимо следить.



Рис. 2.2. Пример конфигурационного файла сервиса

По мимо конфигурационного файла необходимо создать в той же папке файл «cpu.txt», которой будет содержать имя интерфейса и распределение irq\_cores / working\_cores.



Рис. 2.3. Пример задания распределения irq\_cores, working\_cores для сетевых интерфейсов

Для того чтобы можно было использовать флаг –t, который позволяет модулю считывать данные о прерываниях не из файла /proc/interrupts, необходимо создать файл «cpu\_distr.txt».

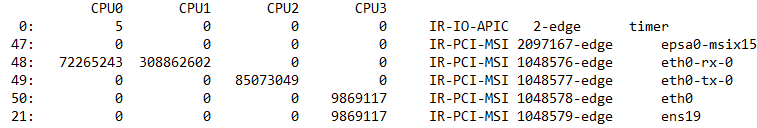


Рис. 2.4. Структура файла с прерываниями

Пути до всех файлов являются абсолютными. При этом указывается лишь директория.

После того как модуль будет успешно запущен, его работу можно будет отслеживать в специальном лог-файле, который будет расположен по пути, указанном при старте ПМ. Если ПМ был запущен в интерактивном режиме, то информация о его работе будет выводить в терминал оператора.

Для того чтобы завершить работу модуля, который работает в фоновом режиме, оператору необходимо посмотреть PID процесса в файле, который располагается в той же папке, где и все остальные конфигурационные файлы и выполнить команду «kill -2 <PID>», с помощью данной команды процессу отправляется сигнал завершения работы. Если модуль был запущен в интерактивном режиме оператору достаточно нажать сочетание клавиш «CTRL+C» и модуль завершит работу.

# СООБЩЕНИЯ

В таблице будут приведены основные сообщения, которые могут появляться в течении работы модуля.

|  |  |
| --- | --- |
| Сообщение | Описание |
| Invalid arguments: path to config files was not specified | Оператор при запуске модуля не указал путь до конфигурационных файлов |
| Invalid arguments: option '-t' requires an argument | Оператор при запуске модуля указал флаг –t, но не указал путь до файла с прерываниями |
| Invalid arguments: dry-run mode can be used only in interactive mode | Оператор при запуске указал флаг -d, но не указал -f |
| Usage: ./vpn\_tune\_service [-f [-d]] [-t <file>] path | Оператор при запуске допустил ошибку в использовании каких-либо флагов, оно показывает пример использования модуля |
| Error: interval value is incorrect | Данное сообщение возникает, если оператор указал неверный формат интервала работы модуля |
| write 15 to /proc/irq/20/smp\_affinity | Данное сообщение показывает, что была осуществлена запись маски равной 15 в файл по расположению /proc/irq/20/smp\_affinity |
| command: echo 15 > /proc/irq/20/smp\_affinity | Данное сообщение показывает, что была сформирована команда для записи маски равной 15 в файл по расположению /proc/irq/20/smp\_affinity |
| cpu\_distribution parameters are: %s" " (netif/irq/work count %d/%u/%u)" " (present/used count %u/%u) | Данное сообщение показывает параметры распределения ядер |